Using speculative execution to automatically hide I/O latency

Fay W. Chang

December 7, 2001

CMU-CS-01-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by graduate fellowships from the Office of Naval Research and Intel, and by generous contributions from the member companies of the Parallel Data Consortium, including 3Com, Compaq, Data General, EMC, Hewlett-Packard, IBM, Intel, LSI Logic, Novell, Quantum, Seagate, StorageTek and Wind River Systems. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of these entities or of the U.S. Government.

Report Documentation Page					*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **07 DEC 2001**	2. REPORT TYPE	3. DATES COVERED **00-00-2001 to 00-00-2001**
4. TITLE AND SUBTITLE **Using speculative execution to automatically hide I/O latency**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**187**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Keywords: Input/output, prefetching, speculative execution, binary modification

Abstract

The gap between processing speeds and disk access times is widening. This trend is causing applications that must fetch data from disk to spend an increasing proportion of their execution times stalled on disk I/O. I/O prefetching, a well-known technique for hiding disk latency, has the potential to alleviate this problem, particularly when the data that needs to be fetched is distributed across multiple disks. A major hurdle to benefiting from this technique in practice is the difficulty of generating accurate and timely prefetches. In this dissertation, I put forth a new approach to generating accurate and timely prefetches without programmer involvement.

The key to the proposed approach is its unique method for predicting what data an executing process will access in the future. The approach involves adding an execution of each target process's code that exploits spare processing cycles. These added executions skip some operations, like accesses to uncached data, so that they can run ahead of their target normal executions. This permits differences between the data values used during the added speculative executions and their target normal executions. Despite any such differences, the approach predicts that the data accesses encountered during speculative executions will often be the same as the data accesses that will be encountered during their target normal executions such that, by initiating prefetching of that data, speculative executions could reduce the I/O stall time of their target normal executions.

To investigate the viability of this approach, I developed and evaluated SpecHint, a design and implementation for applying the approach automatically. SpecHint is based on binary modification and requires no operating system support specific to this approach. I evaluated SpecHint using six benchmarks from the TIP benchmark suite. Experiments demonstrate that, with the file system striped across four disks, SpecHint reduces the elapsed times of five of the benchmarks (text search, scientific visualization, object linking, and two database queries) by 24% to 71%, with an average of 53%. Moreover, simulation experiments suggest that, on future systems, the benefit of SpecHint would not decrease, and may even increase for some applications.

Acknowledgements

I would like to thank my advisor, Garth Gibson, for many years of excellent advice, timely encouragement, and the many experiences that he helped make possible. I would also like to thank the other members of my committee, Greg Ganger, Jim Larus and Todd Mowry, for their helpful comments and questions. Thanks also to Steve Lucco, David Nagle, and Thomas Gross for their advice in earlier years.

I am grateful to the School of Computer Science community at CMU for providing a friendly and inspiring environment unlike any other I have known. I was also fortunate to have been a member of a wonderful lab group, the Parallel Data Lab. Thanks to all the members of the PDL for their camaraderie over the years. Special thanks to Paul Mazaitis for keeping my testbed up and running, Jim Zelenka for help with Digital Unix, and Hugo Patterson and David Rochberg for help with TIP.

Thanks to my friends for laughter, companionship, encouragement, and perspective, which have helped sustain me throughout this endeavor. And, last but not least, thanks to my parents and siblings, who have been a bedrock of support throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many applications often require data that is too large or too infrequently used to be found in memory. Examples of such applications range from common utility programs (like text search or file archiving tools), to database and data mining applications, to a wide variety of scientific and engineering applications. In order to access their data, these applications often issue large numbers of disk requests. Unfortunately, because disks are much slower than processors, these applications often end up wasting a substantial fraction of their execution times waiting for disk requests to complete. Moreover, since the rate at which disks can satisfy requests is increasing slowly relative to increases in processing speeds, this performance problem is expected to worsen.

One way to address this *I/O gap problem* is *I/O prefetching*. Rather than fetching data from disk on demand (i.e. only after an application has attempted to access the data), I/O prefetching involves fetching data from disk in anticipation of an upcoming attempt to access the data. The potential benefit of an I/O prefetch is that, if the anticipated attempt occurs, then the prefetch will have hidden some or all of the time it takes to fetch that data from disk, so the accessing application will waste less time stalled on I/O.

Prior research has demonstrated that, by reducing I/O stall time, I/O prefetching can dramatically improve the performance of a wide range of I/O-intensive applications. In practice, however, it can be difficult to extract such improvements. On one hand, I/O prefetching will yield little or no benefit if too little data is prefetched, or data is prefetched too late, to hide much disk fetch time. On the other hand, if the wrong data is prefetched, or data is prefetched prematurely, then I/O prefetching can degrade rather than improve performance. In particular, since I/O prefetching consumes disk bandwidth, a prefetch could delay other disk requests. Also, since I/O prefetching consumes memory to buffer the prefetched data, a prefetch could cause useful data to be evicted from memory prematurely, so that additional I/O requests are required to fetch that data back into memory. Therefore, prefetching effectively requires both accurate and timely identification of what data will be accessed when.

One approach to obtaining accurate and timely prefetching information is for programmers to implement or modify applications such that, when executed, the applications will issue prefetching calls specifying this information. However, this manual approach can require a substantial amount of programming and debugging effort. Furthermore, a programmer could modify an existing application only if the application's source code is available. Consequently, in practice, very few applications are implemented or subsequently modified to provide such information.

Therefore, a more realistic approach would be to somehow extract prefetching information automatically. Unfortunately, while a variety of automatic approaches have been proposed, there are many applications that cannot be handled by any of the existing approaches. Existing approaches fall in one of three categories: pattern-based approaches, history-based approaches, and static analysis approaches. *Common access pattern approaches* generate prefetches according to some fixed set of pre-identified access patterns. These approaches can be very effective for applications whose data accesses conform to one of their pre-identified patterns, but cannot improve the performance of other applications. *History-based approaches* generate prefetches based on observing the sequence of prior data accesses. These approaches may be effective for prefetching a previously observed sequence of data accesses, but will not, for example, be able to generate accurate prefetches for an access sequence that differs greatly from any which have previously been observed. For this and other reasons, history-based approaches tend to be ineffective for applications whose data accesses depend on inputs that usually vary from run to run. Finally, in *static analysis approaches*, a compiler (or other tool) adds prefetching calls to an application based on pre-runtime analysis of the application's source code. These approaches are limited by the compiler's ability to transform an application such that data will be prefetched substantially in advance of when the application will attempt to access it, an ability which is hampered by the dependence of applications on data values that are determined only during their execution. The difficulty of accomplishing this has thus far constrained static analysis approaches to looping array codes.

This dissertation puts forth a new approach to automating the extraction of I/O prefetching information called the *speculative execution approach*. This approach exploits technology trends, which are providing an increasing abundance of processing and memory resources, to avoid many of the disadvantages of previously existing approaches. In particular, unlike common access pattern approaches, it is capable of prefetching arbitrary access sequences. Unlike history-based approaches, it is capable of prefetching access sequences that have never previously been observed. And, unlike static analysis approaches, it does not require any sophisticated pre-runtime analyses.

1.1 The speculative execution approach

An execution is *speculative* if it depends on one or more possibly incorrect data values, and may therefore be inaccurate. Most processors perform speculative execution to more efficiently handle conditional branch instructions and other sources of pipeline stall. This dissertation advocates performing software-level speculative execution to reduce I/O stall time.

Specifically, this dissertation proposes that, to generate I/O prefetches for an executing application, we should add a new execution of that application's code. This added execution should run ahead of the application's *normal* (i.e. non-speculative) execution by skipping some operations, like attempts to access data that is not in memory. Skipping these operations will probably cause the added execution to use some data values that differ from the data values that will be used during the normal execution. The fundamental hypothesis behind the speculative execution approach is that, despite these incorrect data values, the data accesses encountered during the added speculative execution will often be the same as the data accesses that will occur during subsequent normal execution. Thus, whenever the speculative execution encounters an access to data that is not in memory, it may be able to reduce the normal execution's I/O stall time by issuing a prefetch for that data.

For the speculative execution approach to be successful, it is not sufficient for speculative execution to be capable of generating accurate prefetches for the data that will be accessed during subsequent normal execution. It is also necessary that speculative execution generate these prefetches early enough to hide a substantial amount of disk fetch time. In order to accomplish this, speculative executions must be provided with an adequate and timely amount of processing, memory and I/O resources. Therefore, the approach is also based on a second hypothesis that there are/will be sufficient resources on current/future systems for this purpose.

Finally, a number of factors will affect the practicality of the approach. For example, since programmer time is an expensive commodity, there needs to be a way to add speculative executions automatically. As another example, since people are unlikely to use an approach that causes their applications to produce erroneous results or otherwise malfunction, there should be a way to add speculative executions that will not cause such malfunctions.

1.2 Thesis statement

My thesis is that:

- For a wide range of applications, speculative execution of the application's code could be leveraged to accurately predict what data the application will access in the future early enough for I/O prefetching to provide substantial performance benefits.
- For a wide range of applications, such speculative execution could be added automatically in a manner that will not cause the application to malfunction.

1.3 Dissertation roadmap

This dissertation raises and answers several basic questions. Can speculative execution of an application's code generate accurate predictions of what data will be accessed by the application? What limits the ability of speculative execution to generate accurate predictions? What resources are needed by speculative execution to generate these predictions early enough for effective prefetching? Are these resource needs reasonable on current and future systems? How can adding speculating executions change the behavior of a system? Can we automate the addition of speculative executions in a manner that will not cause applications to malfunction? Can such a design for adding speculative executions deliver substantial performance benefits for real-world applications? To what degree can this approach automatically deliver the potential performance benefits of I/O prefetching? And, relative to prior approaches to automating I/O prefetching, what are the advantages and disadvantages of the speculative execution approach?

To answer these questions, the dissertation discusses the advantages and limitations of this approach to automatically generating I/O prefetches. It also discusses the major issues and tradeoffs in developing a design for automatically adding speculative executions. Interlinked with this discussion, the dissertation provides an in-depth investigation of *SpecHint*, a particular design and implementation. Finally, the dissertation describes an evaluation of SpecHint using five, real-world data-intensive applications. This evaluation demonstrates the strengths and weaknesses of the speculative execution approach and the SpecHint design and implementation.

This dissertation demonstrates that we can automate the speculative execution approach in a manner that is guaranteed to cause no malfunctions for a wide range of applications. The discussions argue and the evaluation affirms that speculative execution is capable of extracting accurate I/O prefetching information, and that the resources needed to perform speculative execution should be reasonable on both current and future systems. The dissertation also argues and affirms that the degree to which the speculative execution approach will be successful is highly application-dependent. In particular, it can be limited by application-specific dependencies of normal execution on data that is not available during speculative execution and/or by application-specific resource requirements for effective speculative execution. The evaluation demonstrates that the speculative execution approach, and the SpecHint design and implementation in particular, can produce impressive performance benefits on a variety of real-world applications. In addition, a simulation experiment indicates that, if current technology trends continue, then these benefits should not decrease, and may even increase for some applications, on future systems. The evaluation also demonstrates that the approach can sometimes deliver benefits comparable to those that could be obtained by manually modifying applications to issue prefetch calls.

The rest of this dissertation is organized as follows. In Chapter 2, I describe the trends which motivate this work and provide background material on I/O prefetching. I then describe and contrast my approach to prior approaches to automating I/O prefetching, and prior uses of speculative execution.

In Chapter 3, I describe the speculative execution approach in greater detail, discussing its potential advantages and disadvantages relative to prior approaches. I also describe the factors which can limit the success of this approach and reason that these factors will not prevent the approach from providing substantial benefits in many interesting situations. I then set the framework for the design discussion in the next three chapters. In particular, I discuss the three basic goals that should be shared by any design for adding speculative execution, sketch out a few possible designs, argue that no single design will be ideal in all situations, and introduce the SpecHint design (the design I developed and implemented in the course of this dissertation work in order to investigate the speculative execution approach).

In Chapters 4, 5 and 6, I describe the major design issues related to each of the three basic design goals. Chapter 4 focuses on effectiveness, the ability of a design to extract timely and accurate prefetching information. Chapter 5 focuses on low overhead, where overhead is defined to be an increase in the elapsed time of a normal execution. Chapter 6 focuses on safety, the degree to which a design ensures that it will not cause applications to malfunction. Each chapter is split into two sections. The first section describes the issues that should be considered in developing a design for adding speculative execution. The second section describes how these issues are addressed in the context of the SpecHint design and implementation.

In Chapter 7, I describe my experimental testbed and the benchmarks I use to evaluate the SpecHint design and implementation. This includes a discussion of how much effort was needed to manually modify the benchmark applications to perform explicit prefetching. In Chapter 8, I describe my evaluation of the SpecHint design and implementation.

Chapter 9 concludes the dissertation with a summary of its key contributions. The chapter also includes some possible avenues for further research stemming from this work.

Chapter 2

Background

In this chapter, I describe the problem that motivates this dissertation work, prior attempts to address this problem, and the reasons why further work is required. Section 2.1 begins by quantifying the huge and growing gap between processing speeds and disk access times. It provides background material to explain why the gap is expected to continue growing, and presents a simple example to illustrate how this growing gap creates a serious performance problem. One way to address this *I/O gap problem* is to apply a well-known technique for hiding disk access times called I/O prefetching. Section 2.2 begins by describing how I/O prefetching can improve performance dramatically. It then explains why I/O prefetching is a difficult technique to apply successfully. Next, it describes previous work on I/O prefetching and why that work is not always sufficient. This dissertation explores a new approach to I/O prefetching based on a technique called speculative execution. Section 2.3 describes the ways speculative execution has been used in the past and how they differ fundamentally from the approach explored in this dissertation. Finally, Section 2.4 briefly discusses other ways of addressing the I/O gap problem.

2.1 The I/O gap problem

Purchase a new desktop computer today, and its processor speed will probably range from 1 to 2 GHz.[1]. In contrast, the average time it takes to retrieve a modest amount of data from disk ranges from 5.6 to 12.6 ms.[2] In other words, the average disk access time on a new system will probably be equivalent to more than 6 million processing cycles. Therefore, an application that needs to wait for data to be retrieved from disk will progress much slower than it otherwise could. Even worse, this gap between processing speeds and disk access times is widening. While processor speeds are increasing at an impressive rate of 58% a year [20], disk access rates are only increasing at about 8% a year [18].

[1]This range comes from the offerings at *www.dell.com* and *www.compaq.com* as of December 1, 2001

[2]This range comes from the offerings at *www.seagate.com* and *www.quantum.com* as of December 1, 2001.

To understand why such a huge gap exists, and why it is expected to continue growing, it is helpful to have a basic understanding of how disks work. A disk drive contains a controller, some buffer memory, a stack of disk platters mounted on a central spindle that rotates at some rate (the *spindle speed*), and a set of disk heads (one for each side of each platter) mounted on an actuator assembly. Data is stored on both sides of each platter in concentric tracks, and the actuator assembly can pivot to position a head over any track. When the controller receives a request for some data, it first checks to see whether that data is in the disk's buffer memory. If so, the controller can begin transferring the data to main memory immediately. Otherwise, the time it takes to service the request (*disk access time*) is the sum of a relatively negligible amount of controller overhead time plus the time to pivot the actuator assembly such that a head is positioned over the track where the data resides (*seek time*), the delay before the data begins rotating beneath that head (*rotational latency*), and the time it takes for all of the requested data to pass beneath the head so that it can be read (*transfer time*). Disk transfer rates have been improving at a respectable 40% a year, largely due to 30% per year improvements in linear density (the density of bits on a track) [18]. However, most disk requests are small enough that disk access times are dominated by *positioning times* (the combination of seek times and rotational latency) rather than transfer time. Unfortunately, as just described, positioning times depend on mechanical technology, which is slower and improves much less rapidly than the solid-state technology that makes up processors, memory and most other components of computer systems.

To illustrate how the widening gap between processing speeds and disk access times can affect application performance, consider a hypothetical single-threaded application that, on a current system, spends twice as much time processing as waiting for data to be retrieved from disk. Assume that (in accordance with current trends) the processor speed increases by 58% per year, and the disk access rate increases by 8% per year. Each year, in accordance with Amdahl's Law[3], the application's elapsed time will decrease by smaller amounts as it spends an increasing proportion of its time stalled on I/O. Figure 2.1 shows both the decrease in elapsed time and the fraction of the new elapsed time that will be spent stalled on I/O, for ten years of annual upgrades. Notice that, after only the second yearly upgrade, the application will be spending more than half of its time stalled on I/O and, after only the eighth yearly upgrade, the application will be spending more than 90% of its time stalled on I/O. Moreover, with the eighth yearly upgrade, the application's elapsed time will decrease by only 11%. In contrast, the elapsed time of an application which performs no I/O would decrease by 58% with each year's upgrade. This example highlights how the widening gap between processor and disk access speeds will hamper the performance of applications that need to access data on disk, causing them to be increasingly disk-bound and increasingly incapable of deriving benefit from advances in processor technology.

The simplified example in the last paragraph ignores possible effects of other trends in computing systems. For example, it has been predicted that increasing memory sizes

[3] A generalized restatement of Amdahl's Law is that the performance improvement which can be obtained from some faster mode of execution is limited by the fraction of time that the faster mode is used.

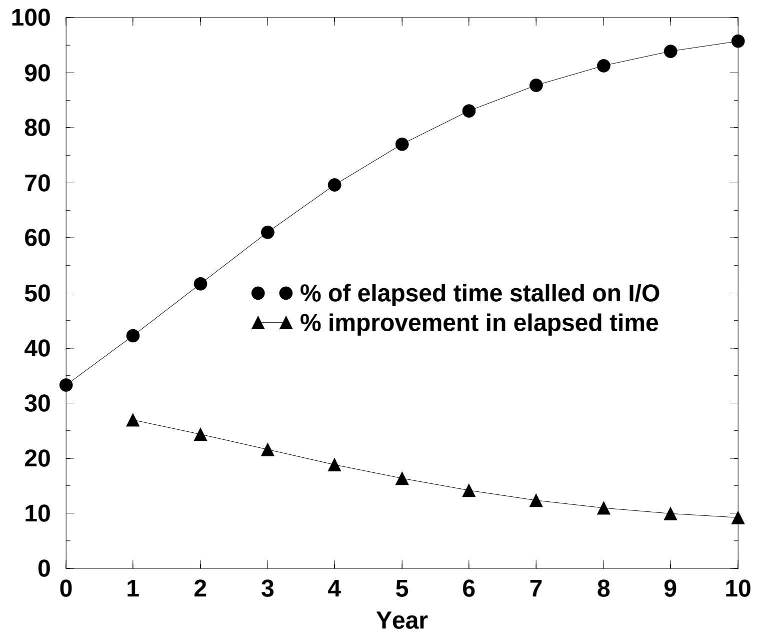

Figure 2.1: Effect of the growing gap between processor speeds and disk access times on the performance of a hypothetical single-threaded application that currently spends twice as much time processing as waiting for disk reads to complete. This graph assumes that, each year, the processor speed increases by 58%, and the disk access rate increases by 8%.

would obviate concern about disk access times by greatly decreasing the percentage of data requests that would require disk access (i.e. the *miss rate*). Several file trace studies have not upheld this predictions, however. In particular, file trace studies from 1985 [39], 1991 [2] and 1999 [65] reported very similar miss rates (approximately 40%), despite the passage of 14 years and the corresponding increase in memory sizes on the traced machines. Although these particular studies were conducted in different computing environments (the BSD, Sprite [38] and NT file systems, respectively), and are not representative of all user communities, they indicate that increasing memory sizes will not automatically hide the performance impact of disk access times. This argues that there will be a growing need for techniques that will explicitly address the I/O gap problem.

2.2 I/O prefetching

Prefetching is a well-known technique for hiding data access latency. Prefetching involves fetching some data from a higher-latency locale to a lower-latency locale (from main memory to processor caches, for example) before the data is accessed. Prefetching can improve performance if prefetched data is subsequently accessed by allowing these accesses to experience only the lower latency. A conceptually simple idea, prefetching is complicated by resource constraints. For example, prefetching consumes bandwidth to transfer the data between the higher and lower latency locales, and buffer space in the lower latency locale to hold the data in anticipation of a subsequent access. Since there are typically competing ways to use these limited resources, prefetching can inadvertently harm rather than help performance.

In this dissertation, I/O prefetching refers to prefetching data from magnetic disks to main memory. Section 2.2.1 shows why I/O prefetching is an appealing technique by describing its potential to produce dramatic performance improvements. Section 2.2.2 shows why I/O prefetching should only be used carefully by describing how I/O prefetching can hurt performance. The challenge with leveraging I/O prefetching is how to extract some of the potential performance improvements described in Section 2.2.1 rather than inadvertently hurting performance as described in Section 2.2.2. Previous research [42, 3] has demonstrated that this problem can be subdivided into two parts: 1) figuring out what data will be accessed when, and 2) assuming the former, figuring out when to prefetch what, and what to eject from memory whenever space is needed (e.g. to hold newly fetched data). This dissertation focuses on the first part of the problem, leveraging prior work that addresses the second part of the problem. Section 2.2.3 summarizes the issues and prior work on addressing the second part of the problem. Section 2.2.4 discusses prior work on the first part of the problem, and why it is not sufficient.

2.2.1 Potential benefits of prefetching

I/O prefetching improves performance by hiding disk access times from applications. In particular, if an application attempts to access some data while it is being prefetched, then the application will experience only the remaining disk access time. Even better, if an application attempts to access some data after it has been prefetched, while the data is still in memory (i.e. still *cached*), then all of the disk access time will be hidden from the application.

To illustrate the extent to which I/O prefetching can improve application performance, consider a hypothetical application that processes for P time units between each of N calls to read data that is not cached when the application begins executing. For simplicity, assume that there are no other executing applications, and that the time it takes for a disk to service an I/O request is fixed at T_{disk} time units.

First, consider the case in which all the application's data resides on a single disk. Each of the application's read calls will trigger an I/O request. Therefore, the application will alternate between processing and stalling on the disk, as shown in Figure 2.2A, and the application's total execution time will be $N(P + T_{disk})$ time units. To determine the potential benefit of prefetching, assume that, as soon as the application begins executing, a prefetch is issued for the data specified by each of the application's N read calls.[4] In this sce-

[4]If an application's data can be prefetched before the application begins executing, then the potential

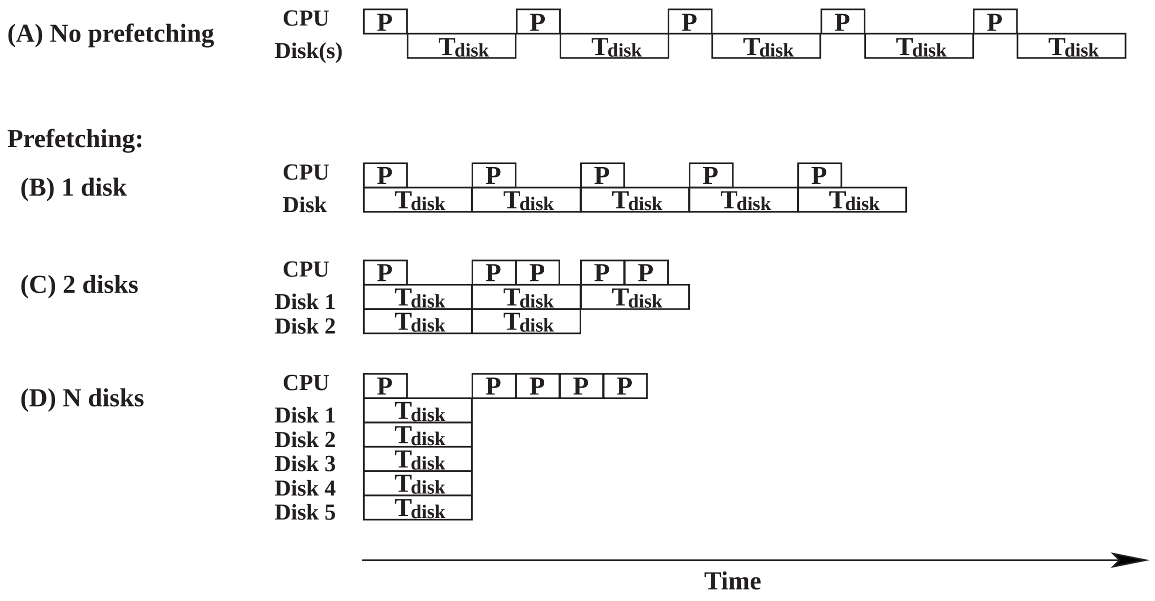

Figure 2.2: This figure illustrates the potential benefit of I/O prefetching. The hypothetical application issues N read calls for uncached data, processing for P cycles between read calls. (A) With no prefetching, the application will take $N(P + T_{disk})$ time units to execute, regardless of the number of disks. To show the potential benefit of prefetching, assume that prefetches are issued for all the data specified in the application's read calls as soon as the application begins executing. (B) With a single disk, the prefetches can be overlapped with computation. (C) With multiple disks, the potential performance improvement is much larger because, in addition to overlapping prefetching with computation, it may be possible to concurrently service multiple prefetches.

nario, when the application begins executing, the disk will begin servicing the first prefetch immediately and will then proceed to service the remaining prefetches one after another. Therefore, as shown in Figure 2.2B, the performance of the application will be improved because the I/O system will fetch the application's data in parallel with the application's processing. Specifically, prefetching will be able to decrease the application's execution time to the greater of NP and NT_{disk}, i.e. by as much as a half.

Even greater performance improvements are possible if an application's data is spread across multiple disks. To illustrate this, reconsider the same application but assume that the application's data is spread across D ($\leq$N) disks such that each call in a sequence of D read calls specifies data on a different disk. If no prefetches are issued, each of the application's read calls will trigger an I/O request. Therefore, the application's execution time will be exactly as it was with a single disk (i.e. Figure 2.2A still applies). If, instead, we assume that prefetches are issued as described in the last paragraph then, when the application begins executing, each of the D disks will begin servicing prefetches. Therefore, as shown

benefit of prefetching may be even greater. This is one advantage of dynamic history-based prefetching approaches, which are discussed in Section 2.2.4.2.

in Figure 2.2C, the I/O system will service D prefetches in parallel with one another, as well as with the the application's processing. This will allow the application's execution time to be decreased to:

$$\begin{cases} N * P & \text{for } P \geq T_{disk} \\ T_{disk} + (N-1) * P & \text{for } T_{disk} \geq P \geq T_{disk}/D \\ \lceil N/D \rceil * T_{disk} + ((D-1) mod N) * P & \text{for } P < T_{disk}/D \end{cases}$$

Therefore, for example, for an application whose data is spread over many disks (i.e. $D \geq 10$) and which performs little processing between read requests (i.e. $P << T_{disk}$), reductions in execution time of 90% or more are plausible.

In addition to showing that the potential benefits of prefetching can be dramatic, the preceding example highlights both how prefetches can allow applications to exploit multi-disk I/O systems more fully and, conversely, how multi-disk I/O systems enable prefetching to be more effective. In particular, since read requests are blocking, a single-threaded application can only have one I/O request outstanding at any time. Moreover, the vast majority of read requests are of a modest size such that the data specified by a single request does not span multiple disks. Therefore, as illustrated in Figure 2.2A, a single-threaded application executing by itself is unlikely to derive substantial performance benefit from having its data distributed across multiple disks because it will never exercise more than one disk at a time. In contrast, prefetching on behalf of an application will increase the frequency with which there are multiple I/O requests outstanding at the same time. If data is well-distributed across multiple disks (where striping [47] generally suffices to produce such a distribution), it is likely that different requests can often be serviced in parallel by multiple disks. Therefore, prefetching increases the degree to which applications can exploit multiple disks.

Conversely, as described above, if an application's processing time between disk requests is less than the disk access time – which, even if not true today, will be increasingly likely given the widening gap between processing speeds and disk access times – then the benefit of prefetching will be limited by the number of disks. For example, from the discussion above, for an application in which P is only 10% of T_{disk}, the maximum potential decrease in the application's execution time is only about 9% if the data resides on a single disk. However, assuming that the application issues 20 read requests, the maximum potential decrease in the application's execution time is 54% if the data resides on two disks, 76% if the data resides on four disks, and 87% if the data resides on ten disks. Notice that prefetching cannot decrease the total amount of data that must be fetched from disk. It can improve performance only by fetching that data earlier than it would otherwise be fetched. Therefore, prefetching may not be able to improve the performance of a multi-threaded application or a multi-tasking system that already fully utilizes the I/O system (i.e. keeps all disks busy at all times) because causing some data to be fetched earlier would simply result in other data being fetched later.

The benefit of prefetching can actually be somewhat greater than described in the preceding discussion. In focusing on how prefetching can hide disk access times from applications, the preceding discussion ignored two ways that prefetching can improve performance by reducing the aggregate time to fetch the necessary data from disk. The first of these leverages the fact that disks can reorder queued requests to decrease their positioning times using any of a variety of disk scheduling algorithms [14]. As discussed above, a typical single-threaded application executing by itself will not benefit from disk scheduling because there will never be more than one request queued at any time. By increasing the depth of disk queues, prefetching increases the potential for a reordering of those requests that will reduce their positioning times. The second of these targets the fact that there is a cost to issuing a request to the I/O system. Since prefetches, unlike demand fetches, are not blocking, it is reasonable to accumulate multiple prefetches, batching them into a single request to the driver in order to reduce driver overhead [43]. Finally, although not considered in this dissertation, I/O prefetching can also improve performance in terms of energy consumption. In particular, by batching I/O requests more closely in time, prefetching can reduce the number of disk spin ups and/or how long disks are powered.

2.2.2 Potential risks of prefetching

While the potential benefits of I/O prefetching are large, prefetching can also hurt application performance as a side effect of increasing contention for a limited resource. This is particularly likely when a prefetch consumes resources without any possible benefit because the prefetched data is not subsequently accessed. There are two primary ways in which a prefetch can hurt application performance.

First, a prefetch can occupy an actuator assembly, delaying demand requests. Current disks are not preemptive; if a demand request is issued to a disk while a prefetch is being serviced, the demand request will not be serviced until the prefetch has completed. Furthermore, current disks cannot differentiate between prefetch requests and demand requests; once a prefetch is issued to a disk, the disk will schedule it no differently than a demand request. If a disk reorders queued requests to reduce aggregate positioning time, it may even reorder a demand request behind a subsequently issued prefetch request. Notice that these hardware limitations make it impossible for any software prefetching scheme to guarantee that it will never hurt performance (unless it can determine when demand fetches will occur, so that it can appropriately limit its activity). One heuristic for bounding the consequences of these hardware limitations is to modify operating systems and device drivers such that they allow a prefetch request to be issued to a disk only when there is at most one request (demand or prefetch) already queued at the disk [43].

Second, since prefetched data competes for space in main memory, applications may end up issuing more I/O requests, and experiencing more I/O latency, to access their data than if no prefetching had occurred. Before a prefetch is issued to disk, space must be reserved in memory to hold the data that will be read from disk. Unless there is sufficient unused memory, reserving this space will require ejecting some data from memory. In addition, until the prefetched data is accessed or ejected from memory, it will occupy space that would otherwise have been available to hold some other data. By shrinking the pool of memory from which space is reserved for subsequently fetched data, prefetched data will indirectly cause the ejection of data that would have otherwise remained in memory. In either case, if the data that is ejected from memory due to a prefetch have been accessed before it was ejected had the prefetch not taken place, then the prefetch will cause that subsequent access to require an I/O that would have been unnecessary had the prefetch not occurred.

2.2.3 Prefetch scheduling and cache management

If an application writer wishes to improve the performance of an application by modifying the application such that it issues I/O prefetches for the data it will subsequently access, the mechanisms offered by current operating systems are poorly suited to the task [42]. In particular, the most suitable mechanism offered by most current operating systems is an asynchronous read call. Asynchronous read calls are a poor prefetching mechanism because issuing such a call causes the operating system to immediately allocate space in memory, and issue a disk request, for any uncached data specified by the call. This presents the application writer with a difficult optimization problem because a particular placement of asynchronous calls in the application which substantially improves performance in one execution environment may be harmful in another. For example, a placement that improves the application's performance when the application is executed on a machine with a certain amount of memory may hurt performance on a machine with less memory because (as described in the previous section) contending for a smaller pool of memory could result in the prefetches causing useful data to be ejected from memory such that the application ends up spending more time stalled on I/O. The same effect could appear even when re-executing the application on the original machine if, during the re-execution, there are concurrently executing applications competing for space in memory. To address this problem, researchers [42, 5] have argued for a separation between exposing what data will be accessed in the future, and allocating shared machine resources in an attempt to derive benefit from that information. Exposing future data accesses will be discussed further in the next section. This section focuses on how a system might manage its I/O and memory resources, given at least partial information about what data will be accessed in the future.

Given some such information, the two main policies that the system can control in striving to extract the greatest benefit from this information are: 1) the prefetch scheduling policy, i.e. when to initiate a prefetch for some data (which typically requires pre-allocating sufficient memory to hold that data before issuing a disk read request), and 2) the cache ejection policy, i.e. what to eject from memory whenever more space needs to be allocated (to satisfy a demand or prefetch request). The challenge in developing a prefetch scheduling policy is how to schedule prefetches such that they are early enough to hide I/O latency entirely, while recognizing when prefetches should be issued later in order to avoid causing ejections from memory that would end up hurting performance. The challenge in developing a cache ejection policy is how to identify what data in memory can be ejected with the least performance penalty.

As discussed above, current operating systems schedule prefetches as if they were demand fetches, i.e. without considering what data they cause to be ejected from memory. The cache ejection policy used by most operating systems approximates the least recently used (LRU) policy. With LRU, when the system needs another cache block, it will eject the data which has not been accessed for the longest time. This policy exploits temporal locality in data accesses (the observation that, in general, data that has been accessed recently is likely to be accessed in the near future) and, while a mismatch for some access patterns, is known to be fairly effective in general. Prior work [43, 3, 4, 24, 62] has proposed a variety of algorithms for leveraging more explicit information about what data will be accessed in the future. Because the system implemented and evaluated in this dissertation takes advantage of TIP2 [43], the rest of this section focuses on TIP2, hereafter referred to simply as "TIP". In particular, Section 2.2.3.1 briefly describes TIP, and then discusses its weaknesses, and how these affect the work described in this dissertation.

2.2.3.1 TIP

The TIP informed prefetching and caching system [43, 41] is an operating system extension that enables processes to indicate what data they will access in the future and, based on this information, attempts to optimize usage of the buffers allotted to the file cache. Specifically, TIP allows each process to generate (by issuing system calls) a single stream of *disclosure hints*, where each disclosure hint specifies some file data. TIP incorporates a system performance model that allows it to estimate, for each hint, the benefit of prefetching the data specified by that hint at any given time. TIP also uses the model to estimate the cost of ejecting data from the *hinted cache*, which consists of the buffers in the cache which contain data that is specified by a hint, and the cost of shrinking the *LRU cache*, which consists of all other buffers in the cache. The latter estimate also depends on a dynamic prediction, derived by observing the stream of file accesses on the system, of how the hit rate of the LRU cache would change if the size of the LRU cache were changed. TIP keeps track of the hint for uncached, unprefetched data that has the largest estimated benefit, and issues the corresponding prefetch once that estimated benefit is larger than the estimated cost of taking the appropriate number of buffers from the hinted cached and/or the LRU cache.

Prior work [41] has demonstrated that TIP can dramatically reduce the execution times of I/O-intensive applications that issue appropriate disclosure hints. For example, it was demonstrated that TIP can reduce the elapsed times of six real-world, I/O-intensive benchmarks executed on a four-disk system by 12% to 72%. However, to obtain these results, it was necessary to manually modify the source code of the benchmark applications so that they would issue appropriate disclosure hints.

Assuming that processes somehow issue disclosure hints, the performance of TIP hinges on the assumptions which underly its system performance model. The most questionable of these assumptions are that: 1) the system supports an infinite amount of I/O parallelism, and 2) a process's hint stream exactly specifies the data that the process will access in the future, in the order that the process will access that data. Two other assumptions in TIP are also questionable. First, TIP assumes that, once prefetched data is no longer specified by any hints, it should simply be added to the tail of the LRU cache's LRU queue. Second, TIP enables processes to cancel hints, but assumes that the underlying system does not allow TIP to cancel prefetches.

TIP's assumption that the system supports an infinite amount of I/O parallelism has several consequences. One is that TIP issues prefetches assuming that prefetch requests will never delay demand requests (which is not the case on current systems for the reasons discussed in Sections 2.2.2 and 2.2.3). In evaluations of the TIP system, this behavior was approximated by a prefetch-aware software striper (described in Section 7.1.1). The assumption of infinite amounts of I/O parallelism also has two consequences that cannot be addressed simply by modifying the systems underneath TIP. First, TIP may not issue prefetches early enough to hide I/O latency completely when there is contention for I/O resources (i.e. if disks have demand requests or other prefetch requests queued). Second, TIP is not capable of exploiting variations in fetch time to hide more I/O latency. For example, assume that data block A is on a lightly loaded disk while data block B is on a heavily loaded disk, so that it would take a much longer time to fetch B from disk than to fetch A from disk. Assume there is a single hint stream, in which A is before B. In this situation, the optimum performance may be achieved by ejecting A to initiate a prefetch for B (possibly initiating a prefetch for A at a later time) because, due to the variation in fetch times, the amount of I/O latency that will be hidden by initiating the prefetch for B earlier is greater than the amount of I/O latency that will be added by re-fetching A at a later time. Since TIP's system performance model does not consider variations in fetch time, it will miss this opportunity.[5]

Unsurprisingly, TIP may make suboptimal prefetching and caching decisions if the hint stream contains incorrect or mis-ordered hints, or is missing hints. In addition, the implementation of TIP makes it possible for incorrect or mis-ordered hints to prevent the process from benefiting from subsequent hints. In particular, there are only two ways to remove a hint from a hint queue; a hint will be removed from its hint queue if the corresponding process issues a matching read request when the hint is at the head of its queue, or if the hint is explicitly cancelled. The estimated benefit of prefetching the data specified by a hint, however, is dependent on the hint's position in its hint queue. Therefore, incorrect or mis-ordered hints need to be explicitly cancelled for the benefit of subsequent hints in the same hint stream to be correctly estimated. If incorrect or mis-ordered hints are not cancelled, TIP will eventually stop prefetching on behalf of the corresponding process.

Adding prefetched data that is no longer hinted to the tail of the LRU queue may not be optimal because the lack of a hint may be an implicit hint that the data will not be reaccessed. For example, consider a fully hinted process that rapidly streams through a large amount of data (i.e. it never re-accesses data). TIP will add this prefetched data to the LRU queue right after the process accesses it. If no other process accesses this data, then the data will consume space in the cache while it moves through the LRU queue, without providing any value.[6]

[5]Subsequent research [24, 62] has proposed solutions to this problem.

[6]Subsequent research [62, 8] has proposed solutions to this problem.

Finally, never revoking prefetches is suboptimal because the estimated benefits and costs of prefetching and ejecting different data from the cache changes dynamically. As a simple example, if a hint is cancelled after a prefetch has been issued for the hinted data, then (as far as the system can determine) there is no benefit to prefetching that data. It may be possible to improve performance by choosing a different use for the cache space pre-allocated for that data, or simply by not wasting I/O bandwidth.

As mentioned earlier, the system that I implemented and evaluated in this dissertation work uses TIP. Notice that these shortcomings of TIP can only cause the benefit of my system to be under-estimated relative to an application that does not provide any information about what data will be accessed in the future. Chapter 5 includes a proposal for how to alleviate the effect of sometimes providing imperfect information (which is evaluated in Chapter 8). TIP's need for explicit cancellation of incorrect and misordered hints is handled by my design and implementation, as described in Chapter 4.

Table 2.1 shows the calls in the TIP interface that were used by my system. In particular, there are two ways to provide a disclosure hint, and a single call which explicitly cancels all hints in the calling process's hint queue. These calls do not produce any application-level side-effects – they do not, for example, update file pointers – so inserting them into an application can affect only the performance of the application (not its output). This increases the usability of the interface because it guarantees that (assuming the implementation of TIP is bug-free) no regression testing (or correctness debugging) will be necessary whenever these calls are added to/moved in/removed from an application. (Of course, after any such change, the programmer may wish to do some performance debugging.)

Ioctl	**Arguments**	**Description**
TIPIO_FD_SEG	file descriptor, offset, length	disclosure hint
TIPIO_SEG	file name, offset, length	disclosure hint
TIPIO_CANCEL_ALL	(none)	cancel all hints queued for this process

Table 2.1: Calls in the TIP interface used by my implementation. These calls do not produce any side-effects (like updating file pointers) that restrict how they should be called in an application. I added TIPIO_CANCEL_ALL to the original TIP interface to increase the efficiency of hint cancellation. (The original interface only allowed the cancellation of a single hint at a time.)

2.2.4 Predicting future data needs

In order to take advantage of any of the algorithms described in the last section, it is necessary to provide information about what data will be accessed in the future. This section discusses prior approaches to extracting such information, and the shortcomings of those approaches.

Perhaps the most obvious approach to exposing this information is to have programmers modify application source code such that it explicitly provides this information, e.g. by issuing the hint calls shown in Table 2.1. To accomplish this, the programmer needs to understand how an application determines what data it will access. The programmer also needs to identify how to modify the application such that it will provide information about what data it will subsequently access early enough for prefetching to hide a substantial amount of I/O latency. For some applications, both of these steps may be easy. Such a case is described in Section 7.3.1. For other applications, however, one or both steps may be difficult. For example, the code which determines what data the application will access may be spread across many modules, and the code may be ordered in a way that requires substantial reorganization before the appropriate information can be provided early enough to derive substantial benefits from prefetching. Such a case is described in Section 7.3.4. Moreover, while the interface through which the programmer exposes this information may not produce any side-effects that could introduce new bugs into the application, reorganizing code may introduce introduce bugs, so such an effort may also require additional regression testing and correctness debugging. Therefore, manual modification has two major drawbacks. First, it can require a formidable amount of programming and debugging effort. Second, it requires that application source code be available. Given that application source code is often not available, and programmer time is an expensive commodity, it is not realistic to expect that all, or even most, applications that could benefit from I/O prefetching will be manually modified to provide timely and accurate information about what data they will access in the future.

The alternative is for this information to be extracted automatically. Prior work on automating the extraction of information about future data accesses can be divided into three categories: common access pattern approaches, dynamic history-based approaches, and approaches based on static analysis. The rest of this section discusses the approaches in each category. The existing approaches in the first two categories were proposed with simple prefetch scheduling policies and no modification of the system's cache ejection policy. One could imagine, however, changing the prefetches to disclosure hint calls, or some other mechanism, to leverage the existing work (described in the previous section) on prefetch scheduling and cache ejection policies. Therefore, the following discussions focus on the strengths and weaknesses of how the different approaches extract information about future data accesses.

2.2.4.1 Common access pattern

A common access pattern approach relies on having previously identified some small set of access patterns that occurs frequently. The basic approach is for the system to observe the last few accesses, check whether these accesses fits one of the previously identified access patterns and, if there is a fit, begin issuing prefetches for the data that will soon be accessed if future accesses continue to fit that pattern.

For example, one of the most widespread forms of automatic I/O prefetching is the sequential file readahead [10, 35] performed by most file systems. Whenever the file system detects that the last n reads to a particular file requested data sequentially (where n varies by file system, but is generally one or two) it begins prefetching whatever data is next, sequentially, in the file (where the amount of data prefetched also varies by file system).

Readahead exploits the preponderance of sequential reads that have been documented in many file trace studies over the years [39, 2, 65]; for example, a recent study [65] found that almost 86% of read and read/write accesses were within a series of sequential accesses. However, readahead is less effective than one might expect from these figures because it can only help performance when the series of sequential accesses spans multiple blocks in the file cache; otherwise, only the first access could suffer any I/O latency. Therefore, it is ineffective when accessing small files. Largely due to accesses to small files, for example, that same recent study included a graph showing that over 90% of sequential read runs were less than 8 KB in size, which is the cache block size in many systems (including Digital Unix 3.2, my evaluation system).

The other widespread form of automatic I/O prefetching is the page clustering [31] performed by most operating systems. Page clustering differs widely from system to system. As an example, consider a virtual memory system that groups pages in an address space into sets of pages called “clusters”. When a page fault occurs, the virtual memory manager generates an demand I/O to fetch the page that was faulted on. With page clustering, it also identifies a cluster to which that page belonged, and generates I/Os to prefetch the other pages in that cluster. One way that page clustering approaches can differ is in how pages are grouped into clusters. If the groups are determined statically, i.e. depending on spatial locality, then the page clustering approach is an example of a common access pattern approach to automating I/O prefetching where the set of pre-identified patterns is spatial locality in memory accesses. If the groups are determined dynamically, i.e. by observing memory accesses, then the page clustering approach is an example of a dynamic history-based approach. Dynamic history-based approaches are discussed in the next section.

Common access pattern approaches have also been proposed for prefetching on behalf of parallel applications. Previous work [26, 34] has identified access patterns that match the accesses performed by many these applications, including complicated patterns like strided accesses.

In general, common access pattern approaches have two major strengths. They can be very effective when the stream of accesses fits one of the pre-identified patterns and, assuming the set of pre-identified patterns is fairly small, they incur little overhead (checking whether the last few accesses fit one of the patterns). However, these approaches have two major weaknesses. First, they cannot help whenever the stream of accesses do not fit any of the pre-identified patterns. Second, they can result in useless prefetching, which can harm system performance as a side effect of increasing resource contention (as discussed in Section 2.2.2), whenever the last few accesses fits a pattern, triggering prefetching, but subsequent accesses do not fit the pattern.

2.2.4.2 Dynamic history

In a dynamic history-based approach to generating I/O prefetches, the system observes the stream of accesses, and possibly some other information. It uses these to form and update a set of prefetching rules, and to determine what data to prefetches (based on those prefetching rules).

One very simple approach [53, 32, 28], sometimes called the "last-successor" approach, is to have the file (virtual memory or database) system track, for each file (page or object), the next file (page or object) accessed the last time that file was accessed. Previous work has shown that this simple approach can correctly predict the next file (page or object) accessed the majority of the time; for example, using traces of file accesses, Kroeger and Long [28] found that this simple approach correctly predicted the next file accessed 72% of the time.

Many more sophisticated approaches have been proposed [60, 40, 6, 16, 32, 28]. Curewitz, Krishnan and Vitter [6] were the first to adapt context modelling techniques generally used for data compression to perform probabilistic prediction of an application's next page fault (or, in an object-oriented database, the next object accessed) based on the sequence of past page faults (or objects accessed). In particular, they used a character-based version of the Lempel-Ziv algorithm and the Finite multi-order context models (FMOC) that originated from prediction-by-partial-match data compressors. Kroeger and Long [27, 28] proposed approaches based on FMOC for predicting the next file that will be accessed. As another example, Griffioen and Appleton [16, 17] proposed an approach in which accesses to files are predicted based on an accumulated graph that tracks, for each file, which files have most often been amongst the next *n* files accessed, where *n* is some small number. This approach differs from the two just described in two related ways: 1) it will implicitly predict more than just the next file that will be accessed, and 2) it will capture the fact that an access to file A is often followed by an access to file B even if there is always an access to some other random file between the accesses to A and B so, for example, it may not be as sensitive to the interleaving of processes on a multi-tasking machine. However, Kroeger and Long [28] found that, while conceptually appealing, such an approach may actually perform worse than the last-successor approach.

Dynamic history-based approaches can be very successful when access streams are repetitive. Approaches that predict file accesses are also capable of discovering and exploiting access patterns that span multiple applications. An example drawn from Griffioen and Appleton's paper [16] is that such an approach could implicitly recognize an edit-compile-run cycle for some source tree and prefetch the appropriate compiler, object files, and/or libraries when it notices that a user is editing a source file in that tree.

On the other hand, dynamic history-based approaches have two critical weaknesses. First, they cannot help when the sequence of accesses is non-repetitive. For example, it is unlikely that a dynamic history-based approach could help an application whose accesses depend on its arguments and inputs if those tend to change each time the application is executed. Second, they are limited by the amount of processing cycles and memory resources they can consume before they incur an overhead that outweighs their benefit. To limit the amount of processing cycles and memory they consume, such approaches must limit the types of events they track, the ways in which they analyze these events, and/or the amount of information they retain about these events. There are many examples of how existing approaches constrain their effectiveness as a by-product of how they limit their overhead. For example, the existing approaches for predicting file accesses track open calls rather than (more numerous) read calls. As a result, they cannot help applications which access large files nonsequentially. A similar example regarding existing approaches for predicting accesses to pages is that, rather than tracking memory accesses, such approaches track only page faults. As a result, the information they gather may become useless if the amount of memory available to an application changes (as would probably occur, for example, if there is a change in what other applications are executing).

2.2.4.3 Static analysis

In a static analysis-based approach to extracting information about what data will be accessed in the future, a compiler analyzes the application at compile-time in order to understand the accesses that the application will make when it is executed. The compiler then transforms the application such that it will provide this information whenever it is executed, e.g. by inserting prefetch calls into the application.

Early work by Trivedi [63] proposed using static analysis to insert prefetch calls, but the proposed analysis applied only to looping array codes that could be tiled (loop tiling [70], also called loop blocking, is a compiler transformation that modifies the way a loop is structured in order to, for example, increase access locality). More recent work by Mowry, Demke and Krieger [36] demonstrated static analysis that could insert prefetch calls into a much broader range of looping array codes. This work also advocates adding a run-time layer which will allow applications to quickly detect which of their pages are in memory. The transformed applications use this run-time layer to avoid the overhead of issuing large numbers of prefetch calls for pages already in memory. This work, and follow-on work by Demke and Mowry [8], also demonstrated static analysis that could insert, into the same kinds of applications, release hints indicating pages that will not be re-accessed before they are ejected from memory. This work demonstrated that, by decreasing memory pressure, correct release hints improve the performance of not only the issuing application, but also any concurrent applications.

Theoretically, approaches based on static analysis could deliver the same performance benefit as having expert programmers modify every application's source code such that the applications will issue prefetch calls. In particular, static analysis tools could potentially perform the same transformation as the expert programmers. In reality, however, these approaches have fallen short of this ideal. The problem is that the static analysis necessary for adding I/O prefetching calls can be prohibitively expensive, in terms of computational cost and/or space. In order to hide the latency of a disk request completely, the compiler should insert a prefetch call such that it will be issued at least an average disk access time earlier than the access. Since disk access times are so large relative to processing times, achieving this requires interprocedural analysis. Many interprocedural static analyses are NP-complete, however [37]. The algorithms that have been developed, therefore, make simplifying assumptions. These assumptions limit the quality of the results they obtain for complex applications. One recent proposal [68] that has leveraged the complexity of static analysis is an approach for protecting trusted software on untrusted hosts based on transforming the trusted software in ways that will make it more difficult to analyze statically. The consequence for I/O prefetching has been that existing static analysis approaches apply only to more structured codes that are easier to analyze – in particular, looping array codes (typically, scientific and engineering applications). On such codes, these approaches have been able to demonstrate impressive results. Most recently, Demke and Mowry [8] demonstrated that, on a four-processor SGI machine with the swap space striped across ten disks, their static analysis approach delivered 16% to 61% reductions in execution time for out-of-core versions of five applications from the NAS Parallel benchmark suite.

2.3 Speculative execution

The approach proposed in this dissertation uses a well-known technique called *speculative execution*. Speculative execution is simply the practice of executing when the execution may be incorrect because it is based on one or more possibly incorrect assumptions about data values. Prior and concurrent work has proposed a variety of different ways in which speculative execution could be leveraged to improve performance.

Most processors perform speculative execution to reduce pipeline stall due to conditional branches. In particular, when a processor encounters a conditional branch, rather than stalling until the branch condition is resolved, the processor predicts whether the condition will indicate that the branch should be taken and continues executing based on this prediction. To ensure that the processor can recover if its prediction turns out to be incorrect, it holds any state changes indicated by the speculative execution in temporary storage until the branch condition is resolved. At that time, if its prediction proves to be incorrect, it discards the uncommitted state changes and executes the correct branch path normally. Otherwise, it commits the pending state changes and continues executing normally, having avoided a pipeline stall.

Researchers have also proposed performing speculative execution to exploit potential parallelism in applications [52, 56, 19]. This idea is sometimes referred to as *thread-level speculation*. Thread-level speculation involves software (language and/or compiler) support for parallelizing code that may contain data dependencies that would ordinarily prevent such a transformation. It also involves processor support for executing chunks of such code in parallel speculatively. In particular, while executing a code chunk speculatively, a processor holds any state changes in temporary storage. It also detects whether it has encountered any data dependencies that may cause it to produce incorrect results. If so, the processor discards the uncommitted state changes and re-executes the chunk. Otherwise, the processor commits the indicated state changes and continues executing, having reduced the application's elapsed time by executing some chunks of the application in parallel.

In the two approaches described above, speculative execution is performed to reduce the amount of normal execution; that is, a speculative execution is considered successful if it proves to be correct, so that execution can proceed normally simply by continuing execution after committing any state changes indicated by the speculative execution. In constrast, my and several other approaches that leverage speculative execution make no attempt to reduce the amount of normal execution. Instead, speculative execution is performed to produce some side effect(s) that have the potential to reduce the time required for normal execution.

For example, Dundas and Mudge [9] proposed that processors perform speculative execution to generate prefetches into the L1 data cache from higher level processor caches and main memory. In particular, they propose that, upon detecting an L1 data cache miss, rather than stalling until the cache miss is serviced, the processor checkpoint the register file to a backup register file, mark the missing data cache word as invalid relative to speculative execution (via an additional tag bit associated with each register and data cache word), and begin executing the subsequent instructions speculatively. During speculative execution, to enable normal execution to be resumed quickly and easily, the processor is permitted to update only register values and the tag bits that indicate whether a register or data cache word is invalid relative to speculative execution. Prefetch requests are issued for memory locations specified by load and store instructions whenever such a location is not formed using any invalid data and is not currently in the L1 data cache. When the original cache miss has been serviced, normal execution resumes at the instruction which caused the original cache miss after simply restoring the register file from the backup register file. This approach could improve performance by reducing the number and/or latency of L1 data cache misses.

More recently, Sundaramoorthy, Purser, and Rotenberg [59] proposed *slipstream processors* that perform speculative execution to produce more accurate branch and value predictions. In their proposal, the operating system instantiates each executed application twice on a slipstreaming chip multiprocessor (or simultaneous multithreaded processor). One of these instantiations performs normal execution. The other performs speculative execution by skipping instructions (and computations leading up to instructions) that the slipstream processors predict will be unreferenced writes, non-modifying writes, or correctly predicted conditional branches. The slipstream processors use the conditional branch directions and the data values calculated during the speculative execution as branch and value predictions during the non-speculative execution. This approach could improve performance by increasing the accuracy of branch and/or value predictions during normal execution.

These two proposals differ fundamentally from mine with regard to what side-effect speculative execution is targetted at producing, and how far ahead speculative execution needs to be, relative to normal execution, in order to produce this side-effect early enough to deliver substantial performance benefit. In these proposals, speculative execution is targetted at producing side-effects that will reduce processor pipeline stall. On current processors, the duration of such a stall is equivalent to at most a few hundred processing cycles. In contrast, my proposal is targetted at producing I/O prefetches that will reduce I/O stall. The duration of an average disk stall, which is equivalent to several million processing cycles, is four or more orders of magnitude longer than a processor pipeline stall. One major ramification of this difference is that, whereas these proposals require maintaining a fairly small amount of additional state information, my proposed approach can require maintaining a substantial amount of additional state information to enable speculative execution to generate correct I/O prefetches that are early enough to hide I/O latencies.

Researchers have proposed many other ways in which speculative execution could be leveraged to improve performance [23]. Unlike my approach, almost all of these approaches, as well as the four approaches discussed above, require special hardware support. The exception, and the prior work closest in spirit to my own, is a proposal by Franaszek, Robinson and Thomasian [12]. In addition to requiring no special hardware support, their proposed approach also uses speculative execution to generate I/O prefetches. More specifically, they proposed that databases be implemented such that, whenever a transaction is unable to execute normally because some necessary lock is not available, the database would be able to execute the transaction speculatively in order to discover and prefetch the data that will be required to perform the transaction. Then, when the locks necessary to perform the transaction become available, the prefetched data would allow the transaction to complete without stalling on I/O. This would minimize the transaction's lock holding time, and therefore the time during which the transaction could prevent other transactions from making progress.

Franaszek, Robinson and Thomasian's proposal differs from my proposal in three critical ways. First, their proposal is directed only at databases. My proposal is directed at a much broader range of applications. One consequence of this difference is that, while they proposed that a transaction be executed speculatively in its entirety before being executed normally, my proposal enables more complex interleavings of speculative and normal execution. Supporting such interleavings raises many issues which did not arise in their work. Second, their proposal was for speculative execution to be included explicitly in the design and implementation of a database. A key part of my work is the claim, and the demonstration, that we can automate the addition of speculative execution for I/O prefetching. This is critical because programmer time is an expensive commodity, and the programming effort necessary to incorporate speculative execution into a system could be considerable. Finally, they evaluated their proposal by performing simulation experiments, and made optimistic assumptions in their simulations. For example, their simulations assumed that speculative execution would always cause the correct data to be prefetched. A key contribution of my dissertation is to demonstrate how successfully speculative execution can generate correct prefetching calls, and also to explore the limitations of speculative execution. In order to accomplish this, I designed, implemented and evaluated a complete working system.

2.4 Other techniques for reducing I/O stall time

This section briefly describes other techniques for reducing the I/O stall time experienced by applications. All of these techniques are complementary to I/O prefetching, and vice versa.

Section 2.2.1 already discussed how disk scheduling algorithms, like CSCAN, can reduce the aggregate I/O positioning time of multiple requests queued at the disk. It also mentioned how batching multiple requests to the driver can reduce driver overhead.

Section 2.2 already discussed the benefit of improving the cache ejection policy, as well as several mechanisms for improving cache ejections if given hints about the data that will (or will not) be accessed in the future. It also mentioned a static analysis approach to adding such hints to looping array codes automatically. Another proposal is to use the name and directory of a file to predict how the file will be accessed and select a cache ejection policy for the file's data [25]. One advantage of improving the cache ejection policy relative to I/O prefetching is that it may also reduce the amount of data that must be fetched from disk. One disadvantage is that it cannot hide the I/O latency of fetching data that is not cached.

Another way to reduce I/O stall time, which may decrease the amount of data that must be fetched from disk, is to reorganize the application's code and/or data to use the cache or disks more effectively by increasing the temporal and/or spatial locality of accesses. One possibility is to use a compiler-based transformation, like tiling [70], which can increase access locality in codes that use regular data structures. Another possibility is to improve the memory allocator. For example, Seidl and Zorn [49] propose using profile information to distinguish between objects based on how frequently they will be accessed and how soon they will be deallocated, then replacing the application's allocator with one that will allocate objects with the similar access frequencies and lifetimes on the same pages. A third possibility is to have a programmer modify an application's source code such that it will issue requests in an order that will incur less disk positioning time.

Rather than reorganizing the application's code or data, another way to reduce I/O stall time is to change the layout of data on disk such that common access patterns, like sequential file accesses, will experience less positioning time. One approach is to improve the file system's disk block allocation policy. For example, since most files are accessed sequentially, many file systems attempt to store a file's data sequentially on disk. The Berkeley Fast File System (FFS) leverages how positioning times, while smallest when consecutive requests are for data that is sequential on disk, are also smaller than average when consecutive requests specify proximate data (e.g. data on the same or adjacent cylinders) rather than data that is further apart. In particular, FFS attempts to store, in proximate cylinders, objects that are likely to be accessed around the same time (e.g. a file's metadata and data) [35]. More recent work proposes embedding file metadata in directories, and storing the data of small files from the same directory in adjacent blocks that are accessed as a unit [13]. A second approach is to rearrange data that is already on-disk. Researchers have explored reducing seek times by rearranging on-disk data such that frequently accessed data is near the middle of the disk [66, 55, 44, 1]. This research exploits the observation that, in general, there is a small subset of the data on a disk which contains the data that will be specified by most of the requests to that disk.

2.5 Summary

The gap between processing speeds and I/O access times is widening. This trend is causing applications that must fetch data from disk to spend an increasing proportion of their execution time stalled on I/O, and to derive diminishing benefits from rapid increases in processor technology. I/O prefetching, a well-known technique for hiding disk latency, has the potential to dramatically decrease I/O stall time, particularly when the data that needs to be fetched is distributed across multiple disks. I/O prefetching can be subdivided into two problems: identifying what data will be accessed in the future, and managing global resources to make the best use of this information. This dissertation work focuses on the first part of the problem, assuming that a reasonable solution to the second problem will be available. In particular, the implementation described and evaluated in this dissertation issues disclosure hints to the TIP prefetching and caching manager.

It is unrealistic to expect that programmers will end up modifying most applications to generate prefetch calls because such modification requires application source code, and can require substantial programming and debugging effort. Therefore, it would be better to have techniques for automating the generation of prefetching calls. Unfortunately, prior automatic techniques are insufficient for generic I/O-intensive applications with non-trivial access patterns. Motivated by these shortcomings, this dissertation work investigates a new automatic approach to generating prefetching calls. This approach is based on speculative execution, a technique that has long been used to avoid pipeline stalls in processors. This dissertation work is the first to explore automating speculative execution in software. It is also the first complete design and evaluation of a system for using speculative execution to hide I/O latency.

Chapter 3

The speculative execution approach and introduction to the design discussion

This chapter introduces the speculative execution approach to automating I/O prefetching and sets the framework for the design discussion in the following three chapters. Section 3.1 begins by describing the speculative execution approach. It then discusses the potential advantages and disadvantages of this approach relative to prior approaches. This discussion suggests that, while there were reasons to conjecture that this approach might be able to deliver substantial performance benefits for a wider range of applications than prior approaches, there were also reasons to believe that the approach would turn out to be impractical.

To resolve this question, I designed, implemented and evaluated a system that applies the speculative execution approach. The second half of this chapter, and the next three chapters, discuss the issues that arise in developing a design for applying the speculative execution approach. Section 3.2 begins by describing the three basic goals for any such design. There are many different designs that could potentially satisfy these goals. The section proceeds to describe a few design possibilities, and argue that no single design will be best in all circumstances. Section 3.2.3 then introduces my dissertation design and implementation, and the assumptions on which it is based. The chapter concludes with a brief summary.

3.1 The speculative execution approach

The speculative execution approach to automating I/O prefetching is a novel approach to initiating I/O prefetching on behalf of target processes in order to reduce the execution times of those target processes. This section begins by describing the approach and using examples to show why it may succeed. Section 3.1.1 then discusses potential advantages of this approach relative to prior approaches. Next, Section 3.1.2 discusses the limitations of this approach, and its disadvantages relative to prior approaches. Finally, Section 3.1.3 discusses why these limitations may not be a problem in many interesting situations.

The speculative execution approach has four components. The first, key component is to add an execution of the code of each target process which strives to predict what data its target process will access, and initiate prefetching according to those predictions. Specifically, the approach involves adding a *speculative execution* of each target process's code that runs ahead of its *target normal execution* (its target process's non-speculative execution) by skipping some operations, like blocking accesses to uncached data. Skipping operations will probably cause the speculative execution to use some incorrect data values, where a data value is incorrect if it differs from the data value that will be used during its target normal execution. For example, if the speculative execution skips a request for file data, then the correct values in that file data will not be available during its subsequent computations. Despite any such differences, the approach predicts that the data accesses encountered during a speculative execution will often be the same as the data accesses that will occur during its target normal execution. Therefore, whenever a speculative execution encounters a data access for uncached data, the approach predicts that the speculative execution may be able to reduce the I/O stall time of its target normal execution by fetching that data into memory. The speculative execution may fetch that data by blocking on the access or, better yet, by issuing a non-blocking prefetch for that data.

To illustrate how this method of predicting what data a process will access can result in accurate prefetches, assume that a speculative execution simply converts every blocking `read` call it encounters into a non-blocking prefetch call for the same data. Now, consider the simple program shown in Figure 3.1. When executed, this program will read `INTSPERFILE` integers from each of the files specified in its arguments, and accumulate the sum of all the integers read. The program will terminate after outputting that sum.

```
void main(int argc, char **argv) {
   int i, j, fd, intbuf[INTSPERFILE], sum;

   for (i=0, sum=0; i < argc-1; i++) {
      fd = open(argv[i+1],"r");
      read(fd, (char *)intbuf, INTSPERFILE*sizeof(int));
      for (j=0; j < INTSPERFILE; j++)
        sum += intbuf[j];
   }
   printf("%d\n", sum);
}
```

Figure 3.1: A simple program used to illustrate how a speculative execution could accurately predict what data a process will access in the future.

Assume that the program is executed with arguments specifying files that are not in memory. When the program's normal execution issues its first read call, the requested data will not be in memory, so it will block on that read call, waiting for the data to be fetched from disk (or a file server). However, the added speculative execution will not be blocked, so it will be able to use this opportunity to run ahead of the program's normal execution. Speculative execution will iterate through the outer `for` loop in `main`, issuing a prefetch call at the beginning of each iteration. Even though it will probably calculate an incorrect value for `sum`, notice that the prefetch calls it issues will accurately specify the data that will be read during normal execution.

The second component of the speculative execution approach is to restrict these added speculative executions to consuming only spare processing cycles. The purpose of this restriction is to prevent the approach from hurting performance by stealing processing cycles from normal executions. Notice that, on a uni-processor, this restriction means that speculative execution should take place only when all normal executions are blocked, including all target normal executions.

The third component of the approach is to ensure that, whenever a speculative execution is actually executing, it will be running ahead of its target normal execution. The purpose of this clause is to prevent speculative executions from hurting performance by stealing memory and I/O resources from normal executions when there is no chance that the speculative executions could provide benefit. In particular, because speculative execution is restricted to consuming spare processing cycles, a speculative execution could easily fall behind its target normal execution. However, a speculative execution can generate predictions only while it is running ahead of its target normal execution, and the sole purpose of performing speculative execution is to generate such predictions. Therefore, if a speculative execution is running behind or (on a multi-processor) in synchrony with its target normal execution, any resources it uses are being wasted.

To illustrate why restricting speculative execution to spare processing cycles may not prevent speculative execution from initiating prefetches that can improve performance substantially, reconsider the above example while focusing on performance rather than prefetch accuracy. Assume that the program is executed on a uni-processor with arguments specifying four files that are stored on different disks, and that no other programs are executing at the same time as this program. As before, assume that the specified files are not in memory and that speculative execution simply converts every read call it encounters into a prefetch for the same data. In addition, assume that, whenever speculative execution is scheduled, it will be *resynchronized* with its target normal execution – that is, the speculative execution's *execution state* (e.g. its program counter, register values, and variable values) will be updated to look like its target normal execution's execution state – so that, as soon as it begins running, it will jump ahead of its target normal execution. Finally, for simplicity, assume that a disk will service a disk request in three million processing cycles, and that `open_files_specified_in_arguments` and the inner `for` loop in `main` each take one million processing cycles to execute. In reality, the average disk access time on a typical current system is equivalent to several million processing cycles, but any given access may take much less, or more, time. Also, the inner `for` loop would probably take much less than one million processing cycles, unless `INTSPERFILE` is extremely large.

Figure 3.2A illustrates how execution would proceed ordinarily (i.e. without speculative execution). The program would execute until it issues its first read call. Since the data specified by this read would not be in memory, execution would block while the data is fetched from disk. Execution will resume only when the disk request completes. It would then proceed only until the second read call since that read call would also specify data

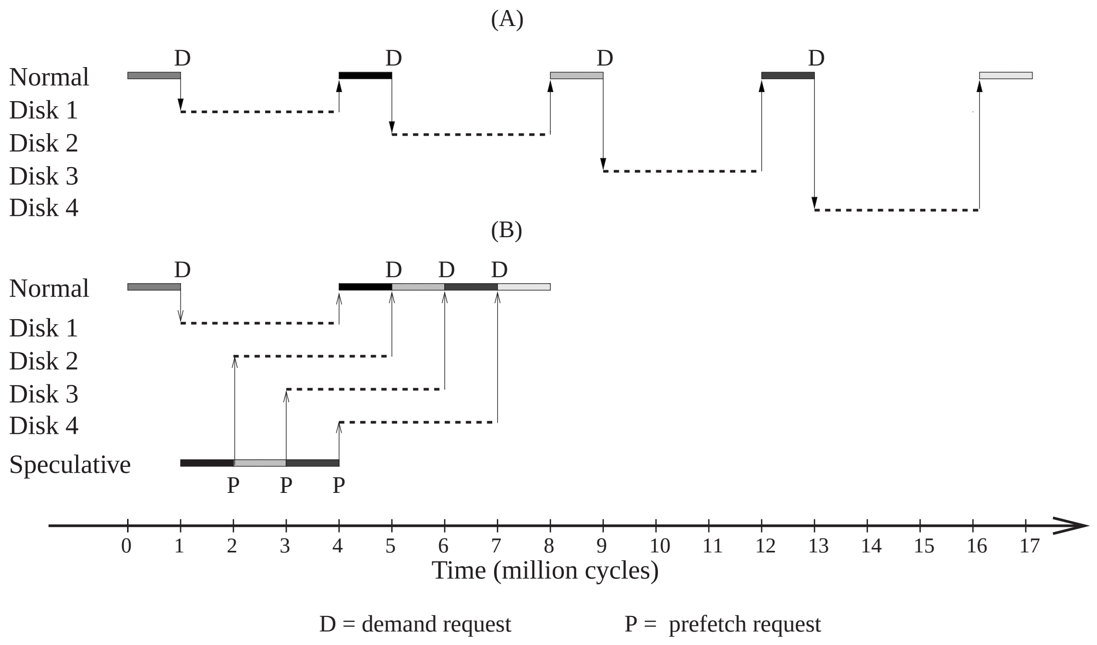

Figure 3.2: Example illustrating how speculative execution could deliver substantial performance benefits. (A) shows how execution would ordinarily proceed for the program shown in Figure 3.1, given the assumptions specified in the text. Basically, execution would alternate between processing and stalling on I/O. (B) shows how execution might proceed for the program if a speculative execution was added. While the normal execution is stalled on I/O, the speculative execution would run ahead of normal execution. Before the initial I/O completes, it would be able to issue prefetch calls for all the data that will subsequently be accessed during normal execution. Therefore, once normal execution resumes, it will experience no more I/O stalls, allowing the program's execution time to be more than halved.

not in memory, causing execution to block again while the appropriate data is fetched from disk. This alternation between processing and stalling on I/O would repeat two more times before the program's execution completes, having taken a total of 17 million processing cycles.

Figure 3.2B depicts how execution would proceed if speculative execution was added. As before, the program would execute until it issues its first read call and then normal execution would block while the data specified by this call is fetched from disk. While normal execution is blocked, the processor would be available for speculative execution. Speculative execution would be resynchronized with normal execution and then scheduled, so that it would pick up from where normal execution is blocked. Before the initial disk request completes, there might be just enough time for speculative execution to iterate through the outer `for` loop three times, issuing three prefetch calls. Once the initial disk request completes, normal execution would resume, preempting speculative execution. Now, however, by the time normal execution issues each of its subsequent read calls, the specified data would have already been prefetched into memory. Therefore, normal execution would experience no more I/O stalls, allowing the program to complete in 8 million processing cycles, i.e. less than half the time that it would take without speculative execution.

The fourth and final component of this approach is to require that speculative executions can be added automatically and safely. In particular, adding speculative execution should not require that a programmer manually modify the source (or binary) code of target applications. Moreover, adding speculative execution should not affect the behavior of a system in a manner that users would consider to be erroneous. For example, returning to the example based on Figure 3.1, the value of `sum` should not be output during speculative execution since users would consider this to be extraneous, incorrect output. This clause is necessary because, without an automatic and safe way to add speculative executions, it is highly unlikely that the approach would ever be adopted.

3.1.1 Potential advantages

This section discusses the potential advantages of the speculative execution approach relative to prior approaches to automating I/O prefetching. As described in Section 2.2.4, the prior approaches each have one or more major weaknesses that cause them to be ineffective for a large set of applications. By not sharing these weaknesses, the speculative execution approach has the potential to greatly expand the set of applications for which I/O prefetching can be automated. In addition, even if a prior approach can automate I/O prefetching for an application, the speculative execution approach may be able to deliver larger performance benefits for that application.

One major weakness of any approach based on pattern matching, whether a common access pattern approach or a dynamic history-based approach, is that its success relies on repetition. In particular, the success of a common access pattern approach relies on access sequences matching one of its fixed set of common access patterns. Therefore, these approaches are ineffective for applications with non-trivial access patterns. Dynamic history-based approaches are more flexible because they can modify their set of patterns based on observing the accesses that occur on the system, but they are still unable to prefetch for access sequences that have not previously been observed. Therefore, for example, they tend to be ineffective for applications whose accesses are input-dependent, assuming those inputs often vary between runs. Because the speculative execution approach predicts future data needs by pre-executing application code rather than by pattern matching, it has the potential to generate prefetches for arbitrary access sequences, including access sequences which have never previously occurred.

Dynamic history-based approaches also have two other major weaknesses. First, to reduce the amount of information they maintain, existing approaches for prefetching file data track only file open calls rather than individual file read calls. Therefore, they are ineffective for applications which access large files non-sequentially. Second, in order to avoid hurting performance by generating many useless prefetches, most existing approaches are designed to prefetch for only the next data access, and the remainder prefetch for only a small number of subsequent data accesses. Therefore, the benefit of these approaches is limited by the time between data accesses, which may be orders of magnitude less than disk access times (e.g. if some of the accessed data is already in memory). The speculative execution approach has the potential to generate prefetches for non-sequential accesses within large files. Moreover, it has the potential to generate prefetches many accesses in advance, which may allow it to deliver larger performance benefits than dynamic history-based approaches.

The main weakness of approaches based on static analysis is that the cost, complexity and precision of the required static analyses increases with the complexity of the code to be analyzed. As a result, all existing approaches are applicable only to simply-structured looping array codes. A secondary weakness is that it can be difficult for these approaches to transform an application in a manner that will generate prefetches efficiently regardless of the values taken on by variables whose values are determined only during the application's execution. The speculative execution approach does not require any sophisticated static analyses. In addition, each time a speculative execution is resynchronized with its target normal execution, it will automatically incorporate any values that were previously determined during that normal execution. Therefore, the approach automatically takes advantage of values that are determined during an application's execution.

3.1.2 Limitations and disadvantages

As discussed in the last section, the two potential advantages of the speculative execution approach are that it may expand the set of applications for which prefetching can be automated, and it may sometimes deliver larger performance benefits than prior approaches to automating I/O prefetching. This section discusses the three main reasons why the speculative execution approach may fail to provide these advantages. It then discusses the disadvantages of this approach relative to prior approaches.

First, the approach may fail whenever the uncached data that will be accessed during a target normal execution depends on data values that are incorrect during speculative execution. When such data dependencies exist, speculative execution may not accurately predict what data will be accessed during its target normal execution. For example, consider the program shown in Figure 3.3. If this program is executed when `SetupFile` and `AlternateSetupFile` are not in memory, then normal execution will block while reading `SetupFile`. Speculative execution will pick up from where normal execution is blocked not knowing the correct value of `mysff.usealternate`. Depending on what value it ends up using, it may or may not issue a prefetch for `AlternateSetupFile`. If it uses the wrong value, it may fail to issue a prefetch that would have improved the performance of its target execution or, even worse, it may issue a useless prefetch which, by consuming I/O and memory resources, could hurt the performance of normal executions.

Second, the approach will fail if there are insufficient memory resources. Immediately after a speculative execution is resynchronized with its target normal execution, their execution states will be identical. As the speculative execution runs ahead of its target normal execution, however, it will probably compute new values for various variables. To reduce

```
struct setupfileformat {
  int usealternate;
  ...
};

void main() {
  int fd;
  struct setupfileformat mysff;

  fd = open("SetupFile","r");
  read(fd, mysff, sizeof(struct setupfileformat);
  if (mysff.usealternate) {
    close(fd);
    fd = open("AlternateSetupFile","r");
    read(fd, mysff, sizeof(struct setupfileformat);
  }
  ...
}
```

Figure 3.3: A simple program used to illustrate how data dependencies can make it difficult for speculative execution to generate accurate prefetches.

the number of incorrect data values it will subsequently use, it may update its execution state to reflect such changes in variable values. This would require additional memory (because a speculative execution should not change the execution state of its target normal execution since such changes might cause the target normal executions to generate incorrect output). Unfortunately, if memory resources are not sufficiently abundant, any memory pressure added by speculative execution could cause useful data to be prematurely evicted from memory, which could severely degrade performance.[1]

Third, the approach will fail if there are insufficient spare processing cycles. In particular, the approach will not be able to improve the performance of a target normal execution substantially if a speculative execution cannot get far enough ahead to discover and initiate prefetching for uncached data substantially in advance of when that data is requested by the target normal execution. There are three reasons why there may be insufficient spare cycles. First, if the target application is multi-threaded or there are concurrently executing applications, then there may be few, or no, spare processing cycles. Next, if the target normal execution consumes a large amount of processing cycles between accesses to uncached data, then, even if there are no other normal executions, there may be insufficient opportunity for a speculative execution to get far enough ahead of that target normal execution. For example, in the example illustrated by Figure 3.2 (of executing the program in Figure 3.1), assume that executing the inner `for` loop required more than three million processing cycles (i.e. more time than servicing a disk request). Then, while normal execution is stalled

[1]To avoid this issue, a particular design for adding speculative executions could constrain the additional memory consumed by speculative execution to some negligible amount. For example, a design could constrain speculative executions such that they could update only register values, or only register and stack values. However, as demonstrated by the results shown in Section 8.2.1, such a design would deliver performance benefits to only a subset of the applications that a less constraining design could benefit.

```
struct fileformat {
  struct _datastats {
    float mean;
    ...
  } datastats;
  float sorteddata[FLOATSPERFILE];
};

void main(int argc, char **argv) {
  int i, j, fd, nbelowmean;
  struct fileformat myff;

  for (i=0; i < argc-1; i++) {
    fd = open(argv[i+1],"r");
    read(fd, myff, sizeof(struct fileformat);
    for (j=0, nbelowmean=0; myff.sorteddata[j] < myff.datastats.mean; j++)
      nbelowmean++;
    printf("%s: %d\n", argv[i+1], nbelowmean);
  }
}
```

Figure 3.4: A simple program used to illustrate how data dependencies can lead a speculative execution to perform a large amount of work that will not be performed during its target normal execution, possibly preventing it from generating useful prefetches.

on the first read call, there would be insufficient cycles for naive speculative execution to complete an iteration of the outer `for` loop, so speculative execution would not be able to initiate any prefetching. Finally, a speculative execution may consume processing cycles performing work that will not be performed during its target normal execution. After having consumed these cycles, there may be insufficient remaining spare cycles. For an example of how this could happen, consider the program shown in Figure 3.4. When executed, this program reads each of the files specified in its arguments. Each file contains an array of sorted integers and some statistics about these integers, including their mean. The program outputs the number of below-mean integers in each file. If this program is executed with arguments specifying files not in memory, then normal execution will block on its first read call. Speculative execution will pick up from where normal execution is blocked not knowing the correct values for the integer data or the mean. Depending on what values it uses, it may iterate through the inner `for` loop either fewer or more times than will its target normal execution. It is possible that it could iterate so many more times that it is unable to start a new iteration of the outer `for` loop and generate a prefetch for the next file before its target normal execution resumes.

These limitations may not be shared by prior approaches to automating I/O prefetching. First, it is possible that a prior approach would be less affected by data dependencies than speculative execution. For example, reconsider the example discussed above that uses the program in Figure 3.3. If `SetupFile` is never modified and indicates that `AlternateSetupFile` should be used, then a history-based approach may be able to consistently prefetch `AlternateSetupFile` as soon as `SetupFile` is accessed. Second, neither common access pattern approaches nor static analysis approaches require a substantial amount of memory, and none of the prior approaches requires spare processing cycles. Therefore, the success of the speculative execution approach is uniquely vulnerable to both the processor and memory utilization of concurrently executing processes. Thus, if one of these other approaches can generate prefetches for an application at least as accurately, abundantly and early as the speculative execution approach, then it is likely that approach will sometimes be able to improve the performance of that application by a larger amount than the speculative execution approach.

The dependence of the speculative execution approach on spare processing cycles actually creates an interesting effect in which speculative executions can be victims of their own success. Basically, the more successful speculative executions are at initiating accurate and timely prefetching, the less frequently their target normal executions will need to block, and therefore the fewer spare processing cycles there will be in which speculative executions can initiate further prefetching. This effect is especially noticeable on a uni-processor system since, on such systems, speculative executions can consume processing cycles only while all normal executions are blocked. For example, if the only accesses a process makes to uncached data are to sequentially read a single large file then, on a uni-processor, it is likely that sequential readahead will be slightly more effective than the speculative execution approach (though the difference may not be noticeable).

In contrast to the situations just described, in which a prior approach may be more effective, recall that there are also factors which would limit the performance benefit that any prefetching approach could deliver. These factors include the percentage of time a process would spend stalled on I/O, the disk locations of the uncached data that a process would access, and the amount of available I/O bandwidth (as discussed in Section 2.2). In addition, data dependencies can make it very unlikely that any automatic prefetching approach would be able to generate accurate prefetches. For example, consider the program shown in Figure 3.5. When this program is executed, it will first read a header from the beginning of the file specified as an argument to the program, and then read some data from that file. Since the location and size of the data specified by the second read call depends on the data specified in the first read call, if the program is executed with an argument specifying a file that is not in memory, it is unlikely that any automatic prefetching approach will be able to generate an accurate prefetch for the second read call.

3.1.3 Discussion

The previous section describes the factors which can cause the speculative execution approach to fail. This section discusses why those factors may not occur so frequently as to make this approach uninteresting.

First, data values that are incorrect during speculative execution can cause the approach to fail for some processes. However, it may be the case that such values will not prevent the approach from succeeding for many processes. One likely scenario in which this should often be true is if the incorrect values are used by some target process only to generate output (as, for example, in the program shown in Figure 3.1). Another likely scenario is

```
struct fileheader {
  int offsetofAdata;
  int sizeofAdata;
  ...
  int offsetofKdata;
  int sizeofKdata;
  ...
};

void main(int argc, char **argv) {
   int fd;
   struct fileheader myfh;
   char *buf;

   fd = open(argv[1],"r");
   read(fd, myfh, sizeof(struct fileheader);
   lseek(fd, myfd.offsetofKdata, SEEK_SET);
   buf = malloc(myfd.sizeofKdata);
   read(fd, buf, myfd.sizeofKdata);
   process_Kdata(buf);
}
```

Figure 3.5: A simple program used to illustrate how data dependencies can make it unlikely that any automatic prefetching approach could generate accurate prefetches.

that the data fetched by a target process during one I/O will determine the next several chunks of uncached data that will be accessed by the target process. For such a target process, speculative execution would not be able to initiate prefetching for all the uncached data, but may be able to initiate prefetching for a substantial subset of that data.

Second, the speculative execution approach will be unsuccessful on any system that is balanced to fully utilize its processing and/or memory resources (e.g. some database installations). However, since the primary performance metric for these systems is often throughput rather than latency, and I/O prefetching cannot improve throughput on a loaded system, employing any sophisticated I/O prefetching approach on these systems would probably be inappropriate.

Third, on most systems, it may be the case that there will often be ample spare processing and memory resources. For example, it has been observed that processing cycles on current systems are often wasted because all applications are blocked waiting for user input, a server response, the network, or disks [46]. Moreover, the rapid advance of processor technology, the increasing popularity of multi-processor systems, and the growth in memory sizes may increase the abundance of spare processing cycles and under-utilized memory.

Finally, it may be the case that the amount of code a target process will execute between accesses to uncached data will usually be reasonably small, so speculative execution will not require a huge number of spare processing cycles to generate predictions and initiate prefetching on behalf of that target process. For example, the second column of Table 3.1 shows the median and average time between read calls for a set of real-world applications on a 233 MHz system. The average disk access time for this system is 15 ms, which is at least an order of magnitude larger than the median time between read calls for all the

Benchmark	**Time between reads**		**Potential prefetches per stall**	
	Median	**Average**	**Median**	**Average**
`Agrep`	126 us	140 us	118	107
`XDataSlice`	952 us	925 us	15	16
`Gnuld`	45 us	131 us	335	114
`Postgres 80%`	910 us	927 us	16	16
`Postgres 20%`	1.1 ms	2.0 ms	13	7
`Sphinx`	83 us	230 ms	181	65

Table 3.1: For a set of benchmarks, this table shows: (in the second column) the median and average time that the benchmark spent processing between read calls, and (in the third column) the number of requests that could potentially be identified during a single I/O stall based on the figures in the second column. The median and average for each benchmark were calculated from the middle 90% of the inter-read times for that benchmark. The number of requests that could potentially be identified during an I/O stall is calculated as $\lfloor 15ms/time\_between\_reads \rfloor$, where 15 ms is the average I/O access time for this system and $time\_between\_reads$ is taken from the second column. The benchmarks and evaluation system are described in Chapter 7.

examined applications. This suggests that a speculative execution of any of these applications may often be able to initiate a prefetch for the next read call that will be issued by its target normal execution using only a small percentage of the cycles during which that target normal execution is stalled on I/O. Moreover, it suggests that, as shown in the last column, if there are no other processes competing for the processor while a target normal execution is stalled on I/O, a speculative execution of these applications may often be able to initiate prefetching for a large number of its target normal execution's subsequent requests.

3.2 Developing a design

This section begins the discussion of how to develop a design for adding speculative execution, setting the framework for the design discussion in the next three chapters. Section 3.2.1 describes the three basic goals for any such design. Each of the next three chapters focus on the design issues relevant to one of these goals. Section 3.2.2 then describes a few possible designs at a very high level, and some potential advantages and disadvantages of each. Section 3.2.3 then introduces the design I developed and implemented in the course of this dissertation work in order to evaluate the potential of the speculative execution approach.

3.2.1 Design goals

Regardless of the particular design, there are three basic design goals:

- *Effectiveness* – applying the design should result in substantial improvements in performance whenever it is possible for the approach to deliver such improvements.
- *Low overhead* – applying the design should never hurt the performance of a normal execution by a noticeable amount; and
- *Safety* – applying the design should never change the output of normal executions in ways users would consider to be incorrect.

The biggest challenge in developing a design for adding speculative executions is trying to achieve all of these goals simultaneously. While the next three chapters each focus on the design issues particularly relevant to one goal, they contain many discussions of design decisions that are tradeoffs between two or all three of these goals.

3.2.2 Design alternatives

This section attempts to give a feel for the range of possible designs by sketching a few possibilities without much detail. It then argues that no single design is "best" by discussing why one or another design might be most appropriate in a given situation.

Consider the following possible designs:

An in-kernel design: This design would be implemented entirely within an operating system. When a target process begins executing, the operating system would fork a child process that performs speculative execution on behalf of that target process. The operating system would mark that child process as a "speculating" process. Whenever the operating system processes a read system call for data that is not in memory, it would check whether the issuing process is a speculating process and, if so, rather than blocking the issuing process until the data is fetched into memory, would initiate prefetching for the specified data and immediately return from the call.

A user-level design based on interpretation: This design would not require any kernel modifications and would be implemented within an interpreter. The interpreter would perform speculative execution by interpreting the code of target processes. Whenever the interpreter encounters a read call, it would issue a non-blocking prefetch call for the same data (using the most appropriate mechanism provided by the operating system).

A user-level design based on binary modification: This design would not require any kernel modifications and would be implemented within a binary modification tool. The tool would accept application binaries and modify the binaries such that, when executed, they would fork a child process that would perform speculative execution on behalf of its parent process. The binaries would be modified such that the child process would issue non-blocking prefetch calls in place of read calls (using the most appropriate mechanism provided by the operating system).

A user-level design based on source code modification: This design would not require any kernel modifications and would be implemented within a compilation system. The compilation system would accept application source code and modify that source code much as described above for a design based on binary modification.

There are many other possibilities – for example, a design that combines some degree of kernel modifications with an interpreter, binary modification tool, or compilation system – but the four above are more than sufficient to demonstrate that no single design will be the best in all situations. For example, a design that requires kernel modifications will not be appropriate for an application-developer or end-user who cannot make modifications in the operating system. On the other hand, all purely user-level designs will be limited by the mechanisms currently supported by the operating system. With these mechanisms, it may be difficult or impossible to achieve the design goals described in Section 3.2.1.

Notice that, even if we restricted ourselves to user-level designs, no single design will be best in all circumstances. Since source code typically contains more information than binaries (source code contains typing, structural and control flow information that is typically not included in binaries), a design based on source code modification may have the potential to leverage the additional information to make speculative execution more effective than interpretation or binary modification designs. However, such a design would not be appropriate when application source code is not available.

3.2.3 The SpecHint design and implementation

For this dissertation, I developed, implemented and evaluated one design in order to demonstrate that an implementation of the speculative execution approach has the potential to substantially improve the performance of a wide range of applications. I chose to develop a user-level design based on binary modification that assumes operating system support for I/O prefetching, but no operating system support specific to speculative execution.

I chose to investigate this particular design point for several reasons. First, I wanted to concentrate on the part of the I/O prefetching problem which was most poorly addressed by prior work – that is, predicting what data will be accessed in the future rather than deciding when to schedule what prefetches. Therefore, I decided to leverage an existing solution to the problem of deciding when to schedule what prefetches – the TIP informed prefetching and caching system (see Section 2.2.3.1). Second, I wanted to develop a design which would be appropriate in the widest range of circumstances, even if not optimal in all of those circumstances. Therefore, I ruled out designs that required operating system support specific to speculative execution and designs based on source code modification. Finally, a design based on interpretation and a design based on binary modification seem appropriate for an equally wide range of situations but, considering the unavoidable overhead of interpretation, it seemed likely that a design based on binary modification could deliver better results.

The design I developed, which I call the *SpecHint design*, is targeted at single-threaded applications and generates prefetches for file data accessed through explicit file read calls.

The most defining design decision in the SpecHint design is that speculative executions take place in the same address space as their target normal executions. In particular, the SpecHint design calls for adding a thread – referred to as a *speculating thread* – to each target process, and using these added speculating threads to perform speculative execution. As discussed in Section 5.2.1, this decision was largely driven by performance characteristics of the operating system targetted by my implementation (Digital Unix 3.2). I mention it here because it had an enormous impact on the rest of the design.

As mentioned above, the SpecHint design assumes operating system support for I/O prefetching. While the implementation assumes the TIP informed prefetching and caching manager, the design is not tied to the TIP manager. In particular, the design could take advantage of any system that provides the following features: 1) prefetching system calls that produce no side-effects (e.g. that do not modify file pointers) so that issuing these calls will always be safe, and that expect data to be specified using a file offset, the number of bytes, and either a file descriptor or a file name, 2) prefetch cancellation calls that allow the retraction of prior prefetch calls, and 3) support for scheduling prefetches that properly accounts for competing demands on memory and I/O bandwidth, so that correct prefetch calls can and should be issued as soon as possible. Notice that the operating system support assumed by the SpecHint design is not specific to speculative execution; any manual or automatic approach to I/O prefetching would be able to take advantage of this support.

My implementation of the SpecHint design, which I refer to as the *SpecHint implementation*, contains three optional elements, not specified by the design. I included these optional elements to investigate various potential optimizations. Chapter 8 evaluates both *base SpecHint* (the implementation of the SpecHint design, without any of these optional elements), as well as the effect of including each of these optional elements.

The discussion of the SpecHint design and implementation is split into three sections (Sections 5.2, 6.3, and 4.2), each of which discusses the design decisions most relevant to one of the three basic design goals (discussed in Section 3.2.1).

3.3 Summary

The speculative execution approach to automating I/O prefetching is a novel approach to initiating I/O prefetching on behalf of target processes in order to reduce their execution times. The key to this approach is the unique mechanism it uses to predict what data a target process will access. In particular, the approach is to add an execution of each target process's code that exploits spare processing cycles. This added execution runs ahead of the target process's normal execution by skipping some operations, like blocking accesses to uncached data. This permits differences between the data values used during the added speculative execution and the data values that will be used during its target normal execution. Despite any such differences, the approach predicts that the data accesses encountered during a speculative execution will often be the same as the data accesses that will occur during its target normal execution. Thus, the approach predicts that, by initiating prefetching for that data, the speculative execution would be able to reduce the I/O stall time of its target normal execution.

The success of the speculative execution approach is based on the following assumptions:

- Given ample processing and memory resources, the uncached data that will be accessed by target processes can often be predicted by a speculative execution of their code.
- There will often be adequate spare processing and memory resources for speculative executions to generate such predictions early enough that data can be prefetched substantially in advance of when it will be accessed.
- It is possible to develop a design for adding speculative executions automatically which is safe and provides an acceptable tradeoff between overhead and performance improvement.

When developing a design for adding speculative execution, there are three basic design goals: safety, low overhead, and effectiveness. There are many possible designs for adding speculative execution that could achieve these goals to varying extents, but no single design will be optimal in all situations. In order to investigate the potential of the speculative execution approach, I developed and implemented a user-level design based on binary modification that assumes operating system support for I/O prefetching, but no operating system support specific to speculative execution. A key element of this design is the addition of a speculating thread to each target process, where speculating threads are responsible for performing speculative execution on behalf of their process's normal execution. The complete description of the SpecHint design is split into three sections, one in each of the following chapters, where each section focuses on the design decisions most influenced by one of the three basic design goals.

Chapter 4

Design goal: Effectiveness

Since the speculative execution approach limits speculative execution to spare processing cycles, and I/O prefetching can improve performance mainly when there is spare I/O bandwidth, designs for adding speculative execution depend on the availability of spare processing and I/O resources. The *effectiveness* of a design for adding speculative execution is how successfully it exploits whatever resources are available on a system to decrease the elapsed time of target normal executions. This is mainly determined by how quickly and accurately the design enables speculative executions to generate prefetches for the uncached data that will be accessed during their target normal executions. This chapter focusses on how a design might increase the speed with which speculative executions will be able to generate accurate prefetches. Section 4.1 discusses several possible methods. Then, Section 4.2 describes the mechanisms in the SpecHint design and implementation.

4.1 Designing for effectiveness

As discussed in Section 3.1.2, incorrect data values and insufficient processing cycles are two of the main factors that could prevent speculative executions from reducing the I/O latency of their target normal executions. Thus, the effectiveness of a design is mainly determined by: 1) what potentially incorrect data values it will cause to be introduced into the execution states of speculative executions, and 2) the speed with which it will enable speculative executions to run ahead of their target normal executions. The first determines whether speculative executions, given an infinite amount of resources, would be capable of generating prefetches that are accurate. The second determines how many processing cycles speculative executions would require to generate prefetches, as well as how early those prefetches would be generated (and, therefore, how much I/O latency they would hide).

A design determines what potentially incorrect data values will be introduced into the execution states of speculative executions in two ways. First, the design determines which operations will differ between speculative executions and their target normal executions. These will include whatever blocking operations (e.g. blocking file read calls) that the design causes speculative executions to replace with non-blocking operations (e.g. non-blocking prefetch calls). It will also include any other operations that the design causes speculative executions to skip (e.g. in order to avoid changing the output of normal executions, as discussed in the previous chapter). Second, given that these differences will cause some data values to be unavailable during speculative execution (e.g. the data values that would have been obtained by a blocking file read call that was replaced with a prefetch call) the design determines what data values speculative execution will use instead during its computations.

A design determines the speed with which speculative executions will run ahead of their target normal executions in two ways. First, the design determines how quickly, and how frequently, a speculative execution will *resynchronize* with its target normal execution (i.e. how quickly and how frequently a speculative execution will update its execution state to match that of its target normal execution, in order to enable it to run ahead of its target normal execution). Second, the design determines the relative speed of a speculative execution and its target normal execution, where a design affects this relative speed by, for example, adding work to speculative execution (e.g. to ensure the safety of the design, as discussed in the previous chapter).

Section 4.1.1 discusses issues related to resynchronizing speculative executions with their target normal executions. Section 4.1.2 discusses a way in which a design might be able to increase the speed of speculative execution. Finally, Section 4.1.3 discusses a way in which a design might increase the performance benefit it delivers by choosing what data values speculative execution uses in place of unavailable data values.

4.1.1 Resynchronizing speculative and normal execution

Recall that the speculative execution approach involves ensuring that, whenever a speculative execution is actually executing, it will be running ahead of its target normal execution. The approach includes this clause in order to prevent speculative executions from wasting resources running behind or, on a multi-processor, in synchrony with their target normal executions (when they would not be able to generate useful prefetches). Thus, a design for adding speculative executions should ensure that, before resuming a speculative execution after it has fallen behind its target normal execution, the speculative execution will first be *resynchronized* with its target normal execution. That is, it should ensure that the speculative execution's execution state (e.g. its program counter, register values, stack, and other data values) is updated to be identical to that of its target normal execution, such that the speculative execution is prepared to run ahead of its target normal execution.

I will refer to the actual process of updating a speculative execution's execution state as a design's *resynchronization method*. For example, in a design based on forking (like the ones sketched in Section 3.2.2), the resynchronization method may simply involve re-forking a new speculating process from the target normal execution's process, and terminating any old speculating process. Notice that a design's resynchronization method should be as efficient as possible because cycles spent resynchronizing will either slow down normal execution (if resynchronizing involves adding work to normal execution), or reduce the number of cycles speculative execution can use to run ahead of normal execution (if resynchronizing involves adding work to speculative execution).

I will refer to a design's choice of when to resynchronize as the design's *resynchronization policy*. Notice that, to properly implement the approach on a multiprocessor, if speculative execution is slower than its normal execution (e.g. because work was added to speculative execution), then a speculative execution that has fallen behind its target normal execution should not be resynchronized and resumed, even if there are spare cycles, until its normal execution blocks. The rest of this section focuses on resynchronization policies for uni-processors, where a speculative execution will only execute while all normal executions are blocked.

One possible policy would be to resynchronize speculative executions with their target normal executions whenever their target normal executions block. Since (on a uni-processor) speculative execution only proceeds when all normal executions are blocked, this policy would ensure that speculative executions will always run ahead of their target normal executions. On the other hand, this policy is pessimistic in that it may resynchronize speculative executions more frequently than necessary, i.e. it may resynchronize a speculative execution that is already ahead of its target normal execution. In particular, consider the following scenario. While a normal execution is blocked, its speculative execution runs ahead, issues a prefetch call for some data, and then continues running even further ahead of the target normal execution. After a while, the target normal execution unblocks, preempting the speculative execution. The target normal execution then requests the data prefetched during speculative execution. If the speculative execution did not issue the prefetch call early enough that the data has already been completely fetched, then the target normal execution will block waiting for the fetch to complete. Notice that, in this case, when the target normal execution blocks, the speculative execution will still be ahead of the target normal execution. Resynchronizing the speculative execution with its target normal execution may simply force it to waste cycles repeating all the work it performed since issuing the prefetch call (plus any cycles spent actually resynchronizing), such that it would issue subsequent prefetch calls later, and may have insufficient cycles to issue as many subsequent prefetch calls, than it otherwise would. (In addition, if it has issued any prefetch calls since the original prefetch, resynchronizing may cause it to re-issue those prefetch calls, which could be problematic on some systems.) Thus, resynchronizing every time a target normal execution blocks may be suboptimal.

On the other hand, a speculative execution that is running ahead of its target normal execution may be so affected by the incorrect data values in its execution state that it will be unable to generate additional correct prefetches until after it is resynchronized again. I will say that such a speculative execution has "strayed". (Notice that, whenever a speculative execution is resynchronized with its target normal execution, it will acquire all runtime values that were previously determined during its target normal execution, such that it will be immune to all prior data dependencies.) Even worse, a speculative execution that has strayed may generate arbitrary numbers of incorrect prefetches, which could hurt performance by increasing contention for memory and I/O bandwidth (as discussed in Section 5.1.4). Therefore, it may sometimes hurt performance to not resynchronize a speculative execution that is running ahead of its target normal execution.

What we would like, therefore, is a mechanism whereby speculative executions will be resynchronized with their target normal executions only when they have either fallen behind their target normal execution, or strayed. This raises the question of how to detect that one of these situations has occurred. One method would be to cross-check some attribute of the *execution paths* taken during speculative and normal execution. For example, assuming the chosen attribute is “procedures called”, if normal execution calls some procedure that was not called during speculative execution, then speculative execution must be either behind normal execution, or on a different execution path (which indicates that it may have strayed).

Being on a different execution path, however, does not always indicate that speculative execution has strayed; differences in execution path are irrelevant unless they would prevent speculative execution from generating correct prefetches. One attribute that would focus on relevant differences in execution paths is the data accessed and identified during normal and speculative execution, respectively. In particular, if normal execution attempts to access some uncached data that was not prefetched by speculative execution, then speculative execution must be either behind, or running ahead but failing to identify all the data that should be prefetched. That is, failed cross-checks of this attribute evince that speculative execution should be resynchronized.

I refer to resynchronization policies based on cross-checking the data accessed during normal execution with the data identified during speculative execution as *lazy resynchronization policies*. Figure 4.1 describes the basic algorithm behind lazy resynchronization policies. Since resynchronization is triggered only upon detecting that speculative execution failed to prefetch some data it should have prefetched, these policies will not always resynchronize speculative execution as soon as speculative execution strays. For the same reason, however, unlike the policy of resynchronizing every time normal execution blocks, these policies avoid resynchronizing speculative execution unnecessarily. Moreover, assuming normal execution only (mainly) blocks to access uncached data, these policies prevent speculative execution from wasting (many) cycles executing behind normal execution.

4.1.2 Skipping unnecessary work

It may be possible for a design to increase the speed with which speculative executions identify the future data needs of their target normal executions. This would increase the effectiveness of the design by enabling speculative executions to generate accurate prefetches earlier and, possibly, to generate additional accurate prefetches within the spare cycles they are allotted. The opportunity exists whenever a target normal execution will perform work that will not affect what uncached data it subsequently accesses – for example, it a target normal execution performs some work solely to generate some final output. Since performing such work during speculative execution would not assist speculative execution to

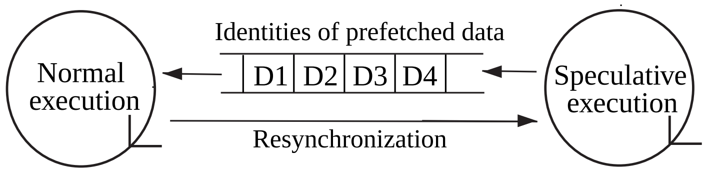

Figure 4.1: Logically, lazy resynchronization policies involve a pipe from speculative execution to normal execution, where the input to the pipe is what data was identified during speculative execution. Normal execution attempts to consume an entry from this pipe whenever it is about to access some data. If the data it is about to access is uncached and either there are no entries in the pipe, or the next entry in the pipe does not identify this uncached data, then it triggers a resynchronization because speculative execution is either behind normal execution, or is running ahead but failing to identify the data it should prefetched.

generate accurate prefetches, a design could increase the speed with which speculative executions generate accurate prefetches by identifying such *unnecessary work*, and causing that work to be skipped during speculative executions.

Such a design might even enable speculative executions to be faster than their target normal executions. On a multiprocessor with an unused processor that could be claimed for speculative execution, this might enable a just-resynchronized speculative execution to outstrip, and therefore generate useful prefetches for, its target normal execution.

A design might, as a simple heuristic, identify and skip work common to many applications that is usually unnecessary. Examples of such work include executing standard library output routines, like `printf`. Uniformly eliding calls to such routines may be only a heuristic because, for example, many of these routines set a return value and/or other data values which might be used during some normal execution. Simple local static analysis may serve to ensure that return values, for example, will not be used during normal execution before eliding some call from speculative execution. In addition, a design might cause speculative execution to skip standard library `assert` calls because speculative execution is likely to fail `assert` calls that normal execution will not fail (since it depends on incorrect data values), and failing such a call would prevent speculative execution from generating more prefetches until it resynchronizes again.

On the other hand, such heuristics will not suffice to identify unnecessary work that is application-specific. One potential approach to identifying application-specific unnecessary work would be to rely on static analysis. In particular, in program analysis parlance, an *executable backwards slice* [69, 61] of a program consists of an executable subset of the program's code that contains all the code that may affect the value of a particular variable at a particular program point. A speculative execution design, therefore, would like to cause speculative execution to skip all code except the combined executable backwards slices of all arguments to system calls that may influence the data requests issued by an application (e.g. `read`, `open`, `lseek`, etcetera). One standard procedure for extracting program slices is to build a *system dependence graph* (SDG) [21] for the program. A SDG captures both the control and data dependencies in an application, allowing a slice to be computed using basic reachability analysis. In order to build a SDG that will yield useful slices, however, good alias analysis is required. While it may be possible to build such a SDG from source code, the state of the art in alias analysis of binaries [7] is not adequate for this purpose. Therefore, unless it requires source code, a speculative execution design cannot rely on existing program slicing technology to identify the unnecessary work that can be skipped during speculative execution.

A design for adding speculative executions, however, is free to employ techniques that may cause speculative execution to diverge further from normal execution (although, to be effective, it should do so only if those techniques would allow speculative executions to hide more I/O latency). Therefore, rather than being limited by existing program slicing techniques, which must be conservative in order to preserve correctness, a design could attempt to identify unnecessary work through experimentation during speculative execution. That is, a design could cause speculative execution to test dynamically whether skipping some work improves or hurts its ability to generate correct prefetches as rapidly as possible. Information gathered during one execution on what work is unnecessary could even be stored and used by future executions [22].

I use the term *experimental slicing* to refer to dynamic testing-based mechanisms for identifying and skipping unnecessary work. To reduce the amount of work that an experimental slicing mechanism adds during speculative execution, a mechanism might include a static component that identifies the chunks of application code that are likely to correspond to substantial amounts of unnecessary work, and are therefore good candidates to test dynamically. The ideal set of such chunks of code would contain all the code that is unnecessary during speculative execution, in the largest (and fewest) possible chunks of code. Larger chunks are desireable because there is likely to be some per-chunk overhead to testing dynamically whether it is beneficial to skip a chunk. On the other hand, depending on how chunks are proposed, since larger chunks contain more code, they may be more likely to include some necessary code.

The dynamic component of an experimental slicing mechanism is determining, through testing during speculative execution, which chunks do not contain any necessary code, and causing these chunks to be skipped in order to increase the speed at which speculative execution generates correct prefetches. Specifically, such a mechanism might perform a series of tests, comparing the speed with which correct prefetches are produced when each chunk is skipped versus not skipped. Ideally, such a mechanism would identify the "I/O kernel" of the application; i.e. the subset of the application code that affects what data will be requested during normal execution. Such a mechanism suggests one way a design for adding speculative executions could easily harness arbitrary numbers of under-utilized processors on a multi-processor; it could perform these tests in parallel.

4.1.3 Increasing the effectiveness of stale values

A speculative execution introduces potentially incorrect data values into its execution state whenever it performs some operation that differs from the operation that would be performed during normal execution. For example, if a speculative execution replaces what would be a blocking file read call for uncached data with a non-blocking prefetch call for that data, the memory buffer into which the data would have been read will probably, during subsequent speculative execution, contain data values that differ from what that buffer will contain during its target normal execution. As discussed in Section 3.1.2, such incorrect data values may prevent speculative executions from generating accurate prefetches (i.e. if the data that their target normal executions will access in the future depend on those incorrect values) and/or decrease the speed with which speculative executions generate accurate prefetches (e.g. by causing speculative executions to iterate through a loop more times than will their target normal executions).

I will refer to values, memory locations or register as “stale” if they may be/contain an incorrect data value. In a simple design, during its computations, speculative executions would use whatever values happen to reside in memory locations and registers, regardless of whether they are stale or how those values were computed. It may be possible, however, that a design could, by causing speculative executions to instead select values for stale registers and memory locations, increase the accuracy of the prefetches they generate and/or the speed with which they generate accurate prefetches. I will refer to values that would enable such increased effectiveness as more “effective” values. Selecting more effective values is similar to value prediction [33] except that the goal of value prediction is to predict the correct value. Figure 4.2 gives a few examples of scenarios in which it may be possible to select a more effective value, including an example of when a more effective value would probably differ from the correct value.

It may also be possible to increase the effectiveness of a design by causing speculative executions to select more effective values for registers and memory locations that are *not* stale. (For example, in Figure 4.2A, even if the correct value of `datastruct[i].num` was known, a zero value would be most effective.) Unlike selecting more effective values for stale registers and memory locations, however, such a mechanism could only increase the set of stale values in a speculative execution’s execution state. Therefore, it could not increase the prefetching accuracy of speculative executions, only their speed, and is just another mechanism through which a design might attempt to skip unnecessary work (in addition to the ones discussed in the previous section). The rest of this section focuses on increasing the effectiveness of stale values.

A design could exploit the fact that is possible to predict the most likely return value for many standard library routines. Consider `read` calls, for example. The parameters to a `read` call specify a file descriptor, the buffer in which the data should be placed, and the number of bytes that should be read. In most cases, the return value from a `read` call will be either the specified number of bytes to read, or the number of bytes from the file pointer associated with the specified file descriptor to the end of the file. Therefore, knowing the file pointer and the length of the file, it would be possible to calculate the likely return

(A)

```
for (i=0; i < NUM_FILES; i++) {
  read(fd[i], datastruct[i], datastructsize);
  for (j=0; j < datastruct[i].num; j++) {
    ... work that does not affect future read requests...
  }
}
```

(B)

```
for (i=0; i < NUM_FILES; i++) {
  fd = open(filename[i], O_RDONLY);
  while (read(fd, buf, bufsize) != 0) {
    ... process data in buf ...
  }
}
```

(C)

```
read(fd, datastruct, datastructsize);
if (datastruct.flag) {
   ... a read request at some offset ...
} else {
   ... a read request at a different offset ...
}
```

Figure 4.2: Examples of code for which it may not be difficult to increase the effectiveness of speculative execution by selecting more effective stale values. In (A), regardless of the correct value for `datastruct[i].num`, a zero value would be most likely to increase the effectivness of speculative execution. In (B), if the return value of the `read` call does not indicate when the end of the file has been reached, speculative execution will not be able to generate prefetches for the next file. In (C), if `datastruct.flag` is not equally likely to be zero or non-zero, then the more likely of these would be a reasonable selection.

value. This example is particularly relevant for speculative execution designs because it is not uncommon for programs to include a loop which issues `read` calls to a file until the return value indicates that the end of the file has been reached (as shown in Figure 4.2B). For such programs, it is necessary to calculate the return values for `read` calls correctly in order to enable speculative execution to both generate prefetches for all the data read from such a file, and then exit the loop such that it can generate additional accurate prefetches. As another example, since most calls are successful, and return values that indicate failure can cause disruptive error handling code to be executed, predicting return values that indicate success may tend to be more effective.

In other cases, it may not be so easy for a design to increase the effectiveness of stale values. A "context-sensitive" approach would be to decide what value to use based on how the value is used. For example, in executing a loop, if a stale value determines the number of loop iterations, then a reasonable heuristic would be to predict a value that would cause the loop to iterate zero or one times. In particular, if the exact number of iterations matters, then it is unlikely that the correct value could be identified but, if the exact number of iterations does not matter, then zero or one iterations would require the least work. Notice, however, that this reasoning only applies if the number of loop iterations is determined by a stale value. If a design can detect through static analysis that the number of loop iterations will be determined by a stale value, then the design may be able to incorporate this heuristic easily. Otherwise, it probably will not be worthwhile for a design to incorporate this heuristic since, without tags on memory locations and registers, keeping track of which values are stale during the course of speculative execution would add a lot of work to speculative execution.

An alternative, "context-insensitive" approach would be to decide what values to use whenever speculative execution introduces new stale values into its execution state. In particular, whenever speculative execution does not issue a system call that would fill some specified memory buffer (e.g. a `read` call), it could select what values to place in that memory buffer. Subsequent speculative execution would propagate the effect of these selected values automatically. This approach does not leverage information about how values will be used, but does not require static analysis or the runtime overhead of tracking which values are stale. Two possible strategies for setting the contents of such input buffers would be to either: 1) pre-select the values with which to fill buffers, or 2) dynamically select values based on the values obtained via prior, similar calls during normal execution. The first strategy assumes that there is some value that is, in general, more effective than whatever values will happen to be in such buffers. The second assumes that, for example, there is similarity between the data obtained from multiple `read` calls that requested the same amount of data (which may be reasonable if, for example, the data is structured).

4.1.4 Scheduling amongst speculative executions

The prior sections have focused on how to improve the effectiveness of individual speculative executions. This section briefly discusses additional issues when multiple speculative executions occur concurrently.

Since speculative execution is strictly unnecessary, it does not make sense to require fairness when dividing available resources amongst speculative executions. A more reasonable goal would be to divide resources in a manner that would yield the greatest combined performance benefit. This may inherently bias against fairness in resource allocation for several reasons. First, speculative executions will vary in how rapidly they generate accurate prefetches and, given that there are a limited number of spare processing cycles, scheduling speculative executions that generate accurate prefetches more rapidly will tend to provide more benefit. Second, if memory resources are not so abundant that the working set of all normal and speculative executions fits in memory, then scheduling a speculative execution whose working set is in memory and can generate accurate prefetches will tend to provide more benefit. Third, if the data required by each execution tends to be physically proximate on disk, which is often the case, then allowing a scheduled speculative execution that is generating accurate prefetches to continue generating accurate prefetches will tend to provide more benefit (by decreasing disk positioning overhead).[1]

This suggests that one reasonable strategy for scheduling amongst multiple speculative executions is to schedule, at some coarse granularity, the speculative execution that would generate accurate prefetches most quickly. If the speed with which any speculative execution generates accurate prefetches tends to stay the same, then this will tend towards job scheduling. A design might predict how quickly each speculative execution would generate accurate prefetches based on: 1) how quickly it has generated accurate prefetches in the recent past, 2) how many accurate prefetches it was able to generate after each resynchronization in the past, and 3) whether the speculative execution has already generated accurate prefetches for the remaining data needs of its target normal execution.

4.2 Effectiveness in SpecHint

This section begins the in-depth description of the SpecHint design and implementation. Recall, from Section 3.2.3, that the SpecHint design is based on binary modification. It assumes a typical UNIX operating system plus support for I/O prefetching, and does not require any operating system modifications. It is targetted at single-threaded applications, and adds a new speculating thread to each target process (which performs speculative execution on behalf of the target process's original thread).

This section describes the mechanisms in the design and implementation that promote effective speculative execution. Section 4.2.1 describes basic support for generating prefetches, Section 4.2.2 describes the resynchronization policy, and Section 4.2.3 describes the resynchronization method. Sections 4.2.4 and 4.2.5 describes an optional experimental slicing mechanism and an optional stale value mechanism in the SpecHint implementation.

4.2.1 Prefetch generation

The speculating thread issues prefetch calls in place of read calls. The design assumes that the operating system provides prefetch system calls that expect data to be specified using a file offset, the number of bytes to prefetch, and either a file descriptor or a file name. Two issues arise in enabling speculating threads to specify the appropriate data in prefetch calls.

First, as further discussed in Chapter 6, the speculating thread is not allowed to issue `open` system calls in order to prevent it from stealing file descriptors from the original thread. Therefore, to enable it to specify the appropriate file in prefetch calls, the speculating thread selects and propagates fake file descriptors rather than issuing `open` system calls, and keeps track of the mapping between these fake descriptors and the file path names that were passed to `open` calls.

Second, unlike `read` and `readv` calls, a prefetch call expects an offset argument. Also, as further discussed in Chapter 6, the speculating thread is not allowed to issue any

[1]This discussion assumes that the I/O system services demand fetches before prefetches, as discussed in Section 5.1.3.

system calls which could change a file pointer (e.g. `lseek`) in order to prevent it from changing the file pointers that will be used during normal execution. Therefore, to enable it to specify the appropriate file offset in prefetch calls, the speculating thread maintains file pointers for itself at user-level. Specifically, the speculating thread obtains the actual offset associated with each (real) file descriptor when it resynchronizes with its original thread, and keeps track of how the offset associated with each (real or fake) file descriptor would change as it runs ahead of its target normal execution (e.g. it updates the appropriate offset rather than issuing an `lseek` system call).

In addition, for the reasons discussed in Section 4.1.3, the speculating thread calculates the probable return values of file read calls, based on its user-level file pointers and file length information that it obtains by issuing `stat` and `fstat` system calls. The speculating thread also uses probable return values that would indicate success for other elided system calls (e.g. it uses its user-level file pointers to select the return value of elided `lseek` system calls), and triggers resynchronization in place of non-returning system calls (i.e. `exit` calls). Finally, for the reasons discussed in Section 4.1.2, the speculating thread skips standard library `assert` calls and, rather than executing standard library output calls (e.g. `printf`, `fwrite`, and `flsbuf`), simply appears to return from these calls with apparently successful return values.

The SpecHint implementation

My implementation of the SpecHint design relies on the TIP informed prefetching and caching system. The prefetch calls supported by TIP do not return any data, even if all of the specified data is in memory. My implementation might be more effective for some applications if the prefetch calls accepted an argument specifying a buffer and placed any portions of the specified data that happened to be in memory in the appropriate positions within that buffer.

4.2.2 Resynchronization policy

The SpecHint design incorporates a lazy resynchronization policy (see Section 4.1.1). In particular, resynchronization is triggered whenever there is a mismatch between the data that the original thread specifies in a file read call, and the data that the speculating thread specified in prefetch calls. Anticipating that files which are opened will subsequently be accessed, resynchronization is also triggered whenever there is a mismatch between the files opened by the original thread and the `open` calls elided by the speculating thread. Finally, resynchronization can be triggered in a variety of other (error-related) circumstances described in Section 6.3, e.g. if the speculating thread triggers an exception.

To implement its lazy resynchronization policy, the speculating and original thread communicate through a circular log (which I refer to as the *prefetch log*), a data structure mapping fake file descriptors to file path names (which I refer to as the *name map*), and a boolean flag (which I refer to as the *resynchronize flag*). (The name map is also used while generating prefetches, as discussed in the previous section.) Whenever the speculating thread encounters what would have been an `open` call, it adds an entry to the prefetch log specifying the fake file descriptor it generates, and updates the name map. Whenever the speculating thread encounters what would have been a file read call, it adds an entry to the prefetch log specifying the (real or fake) file descriptor, offset and length. Whenever the original thread issues an `open` call, it checks whether there is an unconsumed entry in the prefetch log, whether the next unconsumed entry is for an `open` call, and whether that entry indicates the file that the original thread is opening. If so, it keeps track of the mapping from fake file descriptors to real file descriptors; if not, it sets the resynchronize flag. Whenever the original thread is about to issue a file read call, it checks whether there is an unconsumed entry in the prefetch log, whether the next unconsumed entry is for a file read call, and whether that entry indicates the data that the original thread is about to request. If not, it sets the resynchronize flag. The speculating thread polls the resynchronize flag and, upon finding the flag set, executes its part of the resynchronization method (described in the next section).

To enable the original thread to determine efficiently whether an entry specifies the data it is about to read, the original thread keeps track of the current value of its file pointers at user-level (just as does the speculating thread, as described in the previous section). As a final implementation note, to ensure that the speculating thread actually polls the resynchronize flag – i.e. that it cannot get stuck in a loop such that it would be unable to generate additional useful prefetches – the application binary is modified such that the speculating thread will poll the resynchronize flag during each iteration of every loop it executes that could be an infinite or long-running loop.

4.2.3 Resynchronization method

The SpecHint design's resynchronization method involves a trivial amount of work by the original and speculating threads (as indicated by the results in Table 8.8). The design allows resynchronization to occur only while the original thread is blocked on a file read call.

The work performed by the original thread is as follows. Before issuing a file read call, the original thread checks whether the resynchronize flag or the `abort` flag is set. The resynchronize flag is set by the lazy resynchronization policy as described in the previous section, and the abort flag is set whenever resynchronization is triggered by something other than the lazy resynchronization policy, (i.e. by any of the circumstances described in Section 6.3, like the speculating thread triggering an exception). If either of these flags is set, then the original thread saves the values of all its live registers in an array I refer to as the *register store*, and increments a counter I refer to as the *in-read counter*, before issuing the file read call. The live registers include all the callee-saved registers (including the stack pointer), the registers that contain the arguments to the file read call, and its program counter. Upon returning from the file read call, it increments the in-read counter again.

The work performed by the speculating thread is as follows. First, the speculating thread cleans up behind previous speculative execution. In particular, as further described in Section 6.3.2.2, in order to prevent itself from introducing memory leaks, it releases all the memory it has allocated dynamically, re-initializing its data structures appropriately. Also, if resynchronization was triggered because an incorrect prefetch was detected, the speculating thread disowns all prior prefetches. That is, if the resynchronize flag is set, it cancels all unconsumed prefetches, and empties the prefetch log (i.e. changes the value of the variable which stores the index of the last unconsumed entry).

Next, it attempts to update its execution state to be logically identical to that of the original thread. This is complicated by the fact that the original thread could unblock at any time and change its execution state, such that the speculating thread may update its state inconsistently. On the other hand, it is simplified by the fact that the original and speculating threads share an address space, such that the speculating thread's execution state can be made logically identical to that of the original thread simply by updating its program counter, register values, stack, and user-level file pointers (discussed in the last section). The actual process is as follows. First, it loads the value of the in-read counter until that value is odd. An odd value indicates that the original thread is currently blocked on a file read call, such that the values in the register store are consistent with the values in the original thread's stack. Next, it replaces its stack and user-level file pointers with a copy of the original thread's stack and user-level file pointers, and replaces its register values as indicated by the register store. Based on the values in the register store, it also updates the return value register, and its user-level file pointer for the file that the original thread is currently reading, to reflect what they will probably be when that read complete. (As previously discussed, this involves an `fstat` call to obtain the length of the file.) Then, it checks if the value of the in-read counter has changed. If so, then the original thread unblocked while the speculating thread was updating its state, which means that the state the speculating thread copied contains inconsistencies, so the speculating thread repeats this process (i.e. it begins waiting for the value of the in-read counter to become odd again). Otherwise, it knows that its state is consistent, so it begins running ahead of the original thread by jumping to the appropriate address in the code (i.e. after the file read system call on which the original thread is blocked).

To avoid wasting cycles that could be better used by other speculating threads, the speculating thread yields whenever it is waiting for the value of the in-read counter to become odd and the value it loads is even. Notice that this will only occur if the original thread is not blocked on a file read call.

If only the abort flag is set, the speculating thread will not empty the prefetch log. Therefore, after the speculating thread is resynchronized, there may still be valid entries in the prefetch log (i.e. the speculating thread may be behind the furthest ahead it has ever speculated). In this situation, as the speculating thread executes, it will not add entries to the prefetch log, or re-issue prefetches, in place of `open` system calls or file read calls that match the entries in the prefetch log. Furthermore, it will use the fake file descriptors indicated by the entries for `open` calls. If a mismatch occurs, it will cancel the prefetches, and remove the entries, starting from the mismatched entry. At that point, or after running past all the entries in the prefetch log, it will resume writing entries and issuing prefetches.

Finally, for reasons discussed in Section 6.3, the SpecHint design causes there to be, in each transformed application binary, two copies of the code in the original application binary. The "original code" will be executed by the original thread, while its extensively modified copy, which I refer to as "shadow code", will be executed by the speculating thread. Thus, when the speculating thread jumps to the appropriate address to begin running ahead of the original thread, it actually jumps to an address in shadow code. In order to enable the speculating thread to determine the appropriate address in shadow code, the binary modification tool calculates the mapping from the address after each file read system call in original code to the address after the matching elided file read system call in shadow code, and includes an initialized global data structure in the transformed binary that contains these mappings. One further issue is that return addresses are typically stored in the stack, and the return addresses stored in the original thread's stack will indicate addresses in original code rather than shadow code. As further described in Section 6.3.1.4, the design addresses this issue by causing the speculating thread to map return addresses in original code to return addresses in shadow code (once again, using an initialized global data structure included in the transformed binary that contains these mappings).

The SpecHint implementation

As discussed in Section 2.2.3.1, the TIP system allows and requires explicit cancellation of incorrect prefetches. It supports cancellation of either all unconsumed prefetches issued by a process, or the first unconsumed prefetch issued by a process. Therefore, my implementation causes speculating threads to issue a call to cancel all unconsumed prefetches whenever they empty their prefetch log. However, because TIP does not provide a mechanism for cancelling all prefetches after some unconsumed prefetch, my implementation causes speculating threads to simply assume the correctness of any entries in the prefetch log added before the speculating thread last resynchronized.

4.2.4 Experimental slicing

The SpecHint implementation includes, as an optional element, an experimental slicing mechanism. As described in Section 4.1.2, an experimental slicing mechanism enables speculating threads to identify and skip application-specific unnecessary work by dynamically testing whether skipping some work increases the speed at which they generate accurate prefetches. For the experimental slicing mechanism in the SpecHint implementation, the SpecHint tool modifies the binary such that the speculating thread can attempt to skip loops; that is, the SpecHint tool does not perform any static analysis to identify what chunks of code are more likely to correspond to unnecessary work. The rest of this section describes the dynamic component of the implemented mechanism.

To make it easier to interpret the results of a test, chunks are tested one at a time. The more processing cycles the speculating thread spends in a chunk, the greater the benefit will be if the speculating thread is able to skip that chunk. Conversely, testing a chunk in which the speculating thread spends a negligible number of cycles is a risk (because skipping the chunk could prevent the speculating thread from generating accurate prefetches) for little possible reward. Therefore, to determine which chunks to test when, the speculating thread keeps track of roughly how many cycles it spends in each chunk per resynchronization. Non-terminating chunks (i.e. a non-terminating loop), and aborting chunks (i.e. a chunk that triggered an exception) are considered to have taken an infinite number of cycles since they stop the speculating thread from generating additional prefetches until it resynchronizes again. Then, chunks are tested in order from most processing cycles to some threshold amount of cycles per resynchronization (in my implementation, this threshhold is arbitrarily set at 10% of the cycles during an average disk access time).

A test involves resynchronizing and executing while skipping the tested chunk, and then resynchronizing and executing while not skipping the tested chunk. The speculating thread keeps track of the speed at which correct prefetches are produced during each phase of the test and concludes that the tested chunk contains only locally unnecessary code if the speed of the former is substantially greater than the speed of the latter. Only a local conclusion can be drawn because a chunk that is necessary at one point during the execution may be unnecessary at another point, or vice versa. This possibility can be accomodated through retesting, although there is insufficient time during an execution to attempt a full-fledged on-line learning algorithm for noisy, non-stationary environments. On the other hand, there is an overhead to retesting a chunk that has been correctly categorized as necessary or unnecessary because, during the test phase that corresponds to the other category, fewer correct prefetches will be generated. Therefore, the frequency of retests is limited.

Drawing even a local conclusion based on the relative speed at which correct prefetches are produced during the phases of a test can be deceptive since the test phases correspond to different segments of normal execution. Therefore, a difference in the measured speeds may have nothing to do with whether or not the tested chunk is skipped. Instead, it may reflect a difference in the speed at which normal execution could produce reads during these two segments, or a difference in the dependencies on data that are unavailable to the speculating thread during these two segments. A more conclusive test would be to compare the speeds at which correct prefetches are produced while skipping and not skipping the tested chunk over the same segment of the execution. While this would hurt effectiveness on a uni-processor since it would essentially halve the speed of speculative execution, it would be one way to take advantage of an under-utilized multi-processor.

4.2.5 Stale value selection

The SpecHint implementation also includes, as an optional element, a simple context-insensitive stale value selection mechanism (see Section 4.1.3). In particular, if the resynchronization policy has detected that the speculating thread generated incorrect prefetches, then the speculating thread attempts to improve its ability to generate accurate prefetches by logically setting the contents of the buffers that would have been specified in file read calls to a certain value (e.g. to zero). Rather than incurring the cost of actually setting the contents of these buffers, which may involve a substantial amount of work if the buffers are large, the implementation leverages a mechanism specified by the SpecHint design to implement this stale value selection mechanism efficiently. The mechanism it leverages is described in Section 6.3.1.1. For the purposes of this section, the relevant aspect of this mechanism is that its implementation in the SpecHint design requires that the speculating thread execute some code before almost every load and store instruction, and that this code can redirect the subsequent load or store. The implementation leverages this mechanism by allocating some memory, initializing it to contain the selected value, and then redirecting loads from buffers that would have been specified in file read calls since the last resynchronization to instead load from this initialized memory.

4.3 Summary

The effectiveness of a design for adding speculative execution is mainly determined by how quickly and accurately the design enables speculative executions to generate prefetches for the uncached data that will be accessed during their target normal executions. The speed with which speculative executions will generate accurate prefetches is mainly determined by: 1) how quickly and frequently speculative executions are resynchronized with their target normal executions, 2) how much extra work speculative executions perform to support speculative execution, and 3) how much work speculative executions skips because it is unnecessary for generating accurate prefetches. The accuracy with which speculative executions will generate prefetches is mainly determined by: 1) when speculative executions are resynchronized with their target normal executions, and 2) when and what possibly incorrect data values are introduced into speculative execution's execution state.

The chapter discusses three methods in which to increase the effectiveness of a design. It describes a resynchronization policy that avoids resynchronizing unnecessarily. In particular, the lazy resynchronization policy triggers resynchronization only upon detecting that speculative execution is behind, or has failed to prefetch some uncached data that its target normal execution attempts to access. It proposes heuristics and potential mechanisms for enabling speculative execution to skip work that it does not need to perform to generate accurate prefetches. It also proposes heuristics and potential mechanisms for selecting stale values that will increase the effectiveness of speculative execution.

Then, the chapter describes the mechanisms in the SpecHint design and implementation that are focused on increasing effectiveness. In particular, speculating threads will issue prefetch calls in place of read calls, skip unnecessary standard library calls, and use probably correct return values for skipped system and library calls. The design incorporates the lazy resynchronization policy, and an efficient resychronization method. Finally, the chapter describes two optional mechanisms in the SpecHint implementation. An experimental slicing mechanism attempts to enable speculating threads to identify and skip unnecessary work that is application-specific, and a simple stale value selection mechanism attempts to increase the effectiveness of speculating threads by logically filling stale memory buffers with some pre-selected value.

Chapter 5

Design goal: Low overhead

The *overhead* of a design for adding speculative execution is the amount by which it increases the execution times of normal executions. The overhead of a design is important because, even if the design is only sporadically effective, it may still be attractive if it can guarantee that it will never incur a noticeable amount of overhead. Conversely, it seems unlikely that a design will be adopted if, for example, it will sometimes cripple the performance of a system. This chapter focuses on issues related to the design goal of low overhead. Section 5.1 discusses the ways in which a design can incur overhead and the issues with limiting a design's overhead. Section 5.2 discusses some ways in which the SpecHint design was shaped by the goal of low overhead.

5.1 Designing for low overhead

There are two ways that a speculative execution design could hurt the performance of a normal execution. First, it could directly increase the amount of work performed during that normal execution. Second, as a side effect of increasing contention for shared machine resources, it could hurt the performance of that normal execution by disrupting the ability of that normal execution to claim or hold a shared resource.

The first can be avoided by following a simple rule of thumb: whenever possible, a design should ensure that any work it adds is performed during speculative execution rather than normal execution. The SpecHint design described in detail in this dissertation demonstrates that, by following this rule of thumb, it is possible to develop a design that will avoid increasing the work performed during normal execution by a noticeable amount.

The second is harder to avoid. There are three ways in which speculative execution designs can increase resource contention. First, performing speculative execution (and any additional work during normal execution) will consume processing and memory resources, and may consume I/O resources. Second, if speculative execution initiates prefetching, the prefetches will consume memory and I/O resources. Finally, if these prefetches successfully decrease the I/O latency experienced by target normal executions, then the target normal executions will be able to proceed more rapidly, increasing the rate at which they will demand shared resources. In this dissertation, I assume that the performance of target normal executions is of primary importance, such that this last is acceptable. (As an example of how this issue might be addressed, Demke and Mowry [8] have demonstrated an approach that enables prefetching applications to release memory they will no longer need in order to avoid displacing the data of concurrent applications as a consequence of demanding memory more rapidly.) The following subsections discuss the issues that arise in attempting to develop a design that limits overhead as a side effect of increasing contention for shared machine resource by performing speculating execution and issuing prefetches.

5.1.1 Processing cycles

The speculative execution approach calls for restricting speculative execution to consuming spare processing cycles. Most modern operating systems do not provide a mechanism that allows arbitrary executions to be restricted in this fashion. However, on most if not all systems, this restriction can be adequately approximated by setting the scheduling parameters of speculative executions in some obvious way (e.g. by assigning speculative executions the lowest possible scheduling priority). Moreover, since operating systems already support idle threads that execute only during spare processing cycles, if a design is allowed to include kernel modifications, it should be easy to enforce the restriction exactly by modifying the operating system to treat speculative executions as high-priority idle threads.

One other factor should be considered. On some operating systems, there are kernel mode operations which are not preemptible and can take a long time to complete. A design should not allow speculative execution to invoke such an operation. Otherwise, it would be possible for a speculative execution to steal a noticeable number of processing cycles from normal executions.

Forking is one example of an operation that, on some operating systems, is not preemptible and can take a long time to complete. It warrants mention because it is easy to imagine designs that rely on forking. For example, one possible design for adding speculative executions is to fork a child process for each target process and then use the child processes to perform speculative execution. In such a design, an obvious way to help ensure that a speculative execution will run ahead of its target normal execution is for the old child process to be killed and a new child process forked every time the target normal execution blocks. If a design for such a system is allowed to include kernel modifications, then it may be possible to reduce the time it can take to re-fork a child process for the purpose of speculative execution. Otherwise, a design that is not based on forking may be more attractive. As described in Section 5.2.1, the SpecHint design is one example of a design that does not rely on forking.

Finally, notice that restricting speculative execution to use only spare processing cycles is not always optimal; there are scenarios in which prioritizing a speculative execution over a normal execution would improve performance. Consider the example illustrated in Figure 5.1. In this example, a target normal execution blocks when its speculative execution would require only a few more cycles to generate an accurate prefetch for the uncached

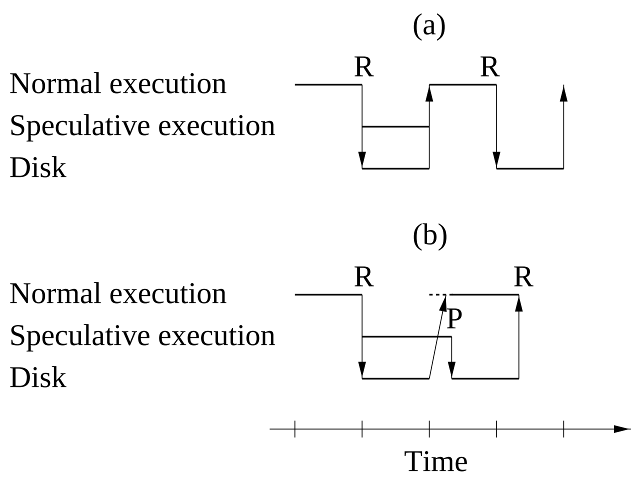

Figure 5.1: One scenario in which, for optimal performance, speculative execution should be prioritized over normal execution. (a) Shows how execution might proceed if speculative execution is preempted by normal execution. (b) Shows how execution might proceed if speculative execution is allowed to continue until it generates a prefetch for the uncached data that will next be requested during normal execution.

data that will next be accessed during the target normal execution. If the normal execution preempts the speculative execution, then the data will still be uncached when the normal execution attempts to access it, such that the normal execution will be forced to stall waiting for that data to be fetched into memory. On the other hand, if the speculative execution is allowed to continue until it has issued the prefetch, then the data will be fetched while the normal execution is processing, such that normal execution will not need to stall when it attempts to access the data.

5.1.2 Memory

Speculative execution will consume memory resources both in order to access the code and data it requires to execute, and to cache the data it causes to be prefetched. The affected memory resources include the processor caches, TLB and main memory. It would be difficult for a speculative execution design to manage its usage of the processor caches and TLB since these resources are hardware controlled. Fortunately, this should not be necessary since the cost of a processor cache or TLB miss is so small (around 100 nanoseconds) relative to the cost of an I/O. (Recall that concurrent normal executions will displace each other's cache and TLB entries regardless of speculative execution, so it is unlikely that speculative execution will greatly increase the number of processor cache or TLB misses.) On the other hand, a design's consumption of main memory could substantially degrade the performance of normal executions by causing them to experience additional I/O stalls.

Limiting the overhead a design can incur through increasing contention for main memory is much more difficult than limiting the overhead it could incur through consuming processing resources. The equivalent to restricting speculative executions to consuming spare processing cycles would be to restrict speculative executions to consuming memory that would otherwise contain only data that could not possibly be re-accessed (e.g. unshared heap and stack data for processes that have terminated). Unfortunately, such a restriction would probably limit speculative executions to allocating so little memory that they would be unable to deliver substantial performance benefits.

On the other hand, there may often be a substantial amount of memory filled with data that could but will not be re-accessed before it is evicted from memory. I will refer to this data as "inactive" data, as opposed to "active" data. To avoid adding overhead by consuming memory, a design would ideally be able to predict whether claiming more memory would cause any active data to be evicted, avoid claiming more memory when this would be the case, and cause data that is in memory only because of speculative execution to be evicted before any active data is evicted.

A design which is not allowed to require operating system modifications will have difficulty even approximating this behavior because the mechanisms provided by most current operating systems are inadequate. For example, most operating systems provide mechanisms only to set hard and soft limits on the amount of memory that can be used by a thread or process (e.g. via the `setrlimit` call), and to obtain information like the per-thread, per-process and system-wide amount of memory being consumed, and the number and rate of page faults and disk requests (e.g. via the `getrusage` and `vmstat` calls).[1] With only these mechanisms, a design would be limited to crude heuristics. For example, a design could set some threshold on how much memory speculative execution can consume, but such a threshold would not protect against when the combined working set of all normal executions is close to the size of memory (such that any additional contention for memory could severely degrade system performance), and could needlessly cripple speculative execution when more memory is available. As another example, a design could attempt to deduce when it is consuming too much memory by observing the rate of page faults and/or disk requests on the system. However, it would not be able to distinguish reliably between when a these occurs because speculative execution is holding too much memory and when they would have occurred regardless of speculative execution. Therefore, a design which is not allowed to require operating system modifications has the poor choice of either risking high overheads, or severely reducing the performance improvements it will deliver in order to be conservative enough to avoid such a risk.

If a design is allowed to include operating system modifications, then better approximations may be possible. For example, the TIP prefetching and caching manager discussed in Section 2.2.3.1 demonstrates how – by observing how frequently the system accesses data that was just evicted, or is nearest to being evicted, from memory – an in-kernel system can estimate the cost of claiming more memory. In addition, a design might modify the opera-

[1]The `man` pages on several operating systems claim that `setrlimit` will allow a process to specify that its memory be reclaimed preferentially if its resident size exceeds some amount. As far as I know, however, only FreeBSD actually provides this functionality.

ting system to keep track of what data is in memory only because of speculative execution, and make that data more likely to be evicted.

5.1.3 I/O bandwidth

A speculative execution design can increase contention for I/O bandwidth in two ways. First, speculative execution will consume I/O bandwidth in order to page in the code and data that it needs to execute, and to prefetch data. Second, speculative execution can cause premature of eviction data that is needed during normal execution, such that additional I/Os will be required to re-fetch that data. Increasing contention for I/O bandwidth can hurt the performance of normal executions in two ways. First, since disks are not preemptive, if a disk is servicing a request on behalf of speculative execution when it receives a request on behalf of normal execution, the normal execution request must wait until the speculative execution request completes, increasing its service time. Second, even if disks were preemptive, simply having moved the disk head in order to service a speculative execution request could increase the service time of the subsequent request.

Ideally, when a design is about to issue a speculative execution request to some disk, it would be able to predict whether a normal execution request will be issued to that disk before the speculative execution request could complete (and the disk arm be returned to its original position), and, if so, delay issuing the speculative execution request. Rather than attempting to predict when normal executions will issue disk requests, one way to approximate this behavior would be to limit the maximum amount by which each normal execution request could be delayed by speculative execution requests. For example, a design could limit the number of speculative execution requests queued for each disk to some small number. In addition, on a system where queued disk requests can be reordered to reduce total positioning time (e.g. most if not all SCSI disks), the design might also require that speculative execution requests be issued only to disks at which no normal execution requests are queued.

If a design is allowed to include operating system support, it should be easy to modify an operating system to support this approximation. For example, as described in Section 7.1.1, the software striper used by the TIP prefetching and caching manager [43] supports something similar.

A design which is not allowed to require operating system modifications would have difficulty supporting this approximation because most operating systems do not expose adequate information. First, since most operating systems hide the location of data, user-level executions cannot detect whether a memory access will trigger a disk request. Therefore, a design may be able to ensure that a speculative execution does not issue a disk request only by blocking the execution. Second, even if a design could always determine when speculative execution is about to issue a disk request, since most operating systems hide the mapping of data to disks, the design would not be able to exploit multi-disk systems fully. In particular, the only way such a design could prevent speculative execution requests from being issued to disks that already have normal execution requests queued would be to delay issuing any requests when any normal executions are blocked on I/O. Since the potential benefits of I/O prefetching are larger on multi-disk systems precisely because multi-disk systems allow prefetches to be serviced in parallel with demand requests, this would severely reduce the performance improvements the design could deliver.

5.1.4 Useless I/O

Since it is difficult to guarantee that consuming memory and/or I/O bandwidth will not hurt the performance of normal execution (for the reasons discussed in the prior two sections), a design should attempt to avoid initiating *useless* I/O, where an I/O is considered useless if it does not improve the performance of a normal execution.

A speculative execution can initiate two types of useless I/O directly: useless demand fetches and useless prefetches. In addition, if the system is paging, it may write some or all of a speculative execution's *private data* (stack, heap and global data that is not shared with any other execution) to swap space, such that a speculative execution could cause useless writes.

A demand fetch initiated by a speculative execution is useless if no normal execution accesses the fetched data before it is evicted from memory, and the speculative execution either issues no more accurate prefetches or would not have needed that data to issue its subsequent accurate prefetches. We know that a speculative execution's private data will not be accessed by a normal execution, so a demand fetch for a speculative execution's private data is useless unless the speculative execution subsequently uses that data to issue accurate prefetches. Therefore, to reduce useless demand fetches, a design might include mechanisms for predicting whether a speculative execution will produce no more accurate prefetches, and preventing such a speculative execution from initiating demand fetches for its private data. On the other hand, if speculative execution initiates a demand fetch for non-private data (i.e. data it shares with another execution), the demand fetch itself may turn out to be a prefetch for a normal execution. For example, in some designs, a speculative execution might issue demand fetches for code pages that may subsequently be used by its target normal execution.

Writes of a speculative execution's private data are likely to prove useless because, as discussed in Section 3.1, the private data of a speculative execution is replaced whenever the speculative execution is resynchronized with its target normal execution. Therefore, a design that is allowed to include operating system modifications might reduce its overhead by not writing the private data of speculative executions to swap space. A design that is not allowed to require operating system modifications will not be able to reduce its overhead in this manner because current operating systems do not provide an appropriate mechanism.

The rest of this section discusses useless prefetches.

5.1.4.1 Useless prefetches

Recall that, whenever a speculative execution (running ahead of its target normal execution) attempts to access some data that is not in memory, using a non-blocking operation in place of what would have been a blocking operation during its target normal execution, the approach considers that data a reasonable candidate for prefetching. For each of these *prefetch candidates*, depending on the particular design, the speculative execution may or may not initiate a prefetch. A design may cause speculative execution to not issue prefetches for some of the prefetch candidates it uncovers in order to avoid hurting performance through useless prefetches. However, if a design causes prefetch candidates to not be prefetched when prefetches would have proven correct, then the design would have sacrificed some performance benefit. Furthermore, since a useless prefetch may incur no overhead if there is ample memory and I/O bandwidth, there are circumstances under which it may be worthwhile to issue a prefetch that seems likely to prove useless on the off-chance that it proves correct. The rest of this section discusses how a design might predict the likelihood that a prefetch will be useless, and when a design might decide whether to issue a prefetch that seems likely to be useless.

Predicting correctness probability

If a design could detect when the data accessed during normal execution will depend on data values that are incorrect during speculative execution, then it could eliminate all useless prefetches without sacrificing any correct prefetches. One possible approach would be to pre-analyze the target application's code. In particular, using static analyses, one could attempt to identify what code might trigger a data access that might depend on data values that might be incorrect during a speculative execution of that code. This approach would probably work well for some simply-structured applications in which the existence, or lack, of dependency between data accesses is easy to detect. However, for complex applications, the imprecision of interprocedural analyses of complex code (as discussed in Section 2.2.4.3) would probably lead to imprecise results – for example, rather than identifying exactly which data accesses depend on incorrect data values, it might identify a large superset of those accesses. Moreover, it might be difficult to extend this approach to take advantage of the fact that the set of incorrect data values could shrink whenever speculative execution is resynchronized with normal execution.

An alternative approach might be to detect whether a prefetch candidate depends on any incorrect values by keeping track of such dependencies as speculative execution makes progress. For example, speculative execution could maintain an incorrectness bit for each register and memory location used during speculative execution, setting a register's (or memory location's) bit whenever its value was changed based on a value that is already marked as being incorrect. One problem with this approach is that it would require many additional instructions to maintain the incorrectness bits. A more critical problem is that it is unclear how speculative executions should handle control dependencies; that is, how should the incorrectness bits be set after speculative execution encounters a conditional branch instruction (an instruction which changes the program counter based on some value) which depends on a value that is marked as being incorrect?

In lieu of these more complicated approaches, a design might simply attempt to predict the probability that it would prove correct to prefetch a new prefetch candidate based on observing whether it would have proven correct to prefetch prior prefetch candidates – i.e. to use a history-based approach to predicting the *correctness probability* of new prefetch candidates. I will refer to a prefetch candidate for which a prefetch would prove useless as a “bad” candidate, and a prefetch candidate for which a prefetch would prove correct as a “good” candidate.

There are many possible history-based approaches. The ideal approach would predict a 100% correctness probability for every good prefetch candidate, and a 0% correctness probability for every bad prefetch candidate. A good approach would predict a high correctness probability for all good prefetch candidates, and a low correctness probability for a large number of bad prefetch candidates, such that it would be easy to eliminate a large number of useless prefetches without reducing the number of correct prefetches. An approach that predicts the same correctness probability for all prefetch candidates would be useless because it would not be helping to distinguish between good and bad candidates. Finally, an approach which predicts a low correctness probability for a substantial number of good prefetch candidates might be worse than useless because it could deceive speculative execution into eliminating a large number of correct prefetches, which could greatly decrease its performance benefit.

A simple history-based approach would be to predict that a new prefetch candidate’s correctness probability is equal to the percentage of the speculative execution’s recent prefetch candidates that were good, i.e. the ratio of good to total candidates during the last t seconds of this speculative execution. This approach would be cheap and easy to implement either in the kernel or at user-level. It would also allow a speculative execution to differentiate between periods of generating mostly good prefetch candidates and periods of generating mostly bad prefetch candidates. However, if a speculative execution generates a mix of good and bad prefetch candidates, and the mix is the same within every t seconds, then this approach will assign every prefetch candidate from that speculative execution the same correctness probability. This might be somewhat useful in allowing the system to differentiate between speculative executions that generate mostly good prefetch candidates and those that generate mostly bad prefetch candidates, but (as discussed in the last paragraph) it would be completely useless if no other speculative executions were taking place.

A more sophisticated approach, which is reminiscent of branch prediction techniques, is to categorize prefetch candidates based on some features of the execution path taken to generate the prefetch candidate. This approach makes sense because, as illustrated in Figure 5.2, the execution path taken to generate a prefetch candidate determines whether that prefetch candidate depended on incorrect data values. A prefetch candidate’s correctness probability can then be predicted as, for example, the percentage of recent prefetch candidates in the same category that were good.

The key design decision for such a *path-based prediction approach* is what set of features to track. There are many possible *feature sets*, e.g. the address of the last read call, the target address of the last *m* control transfers, or the last *n* return addresses on the stack. The ideal feature set would, for all applications, be inexpensive to track and result in two categories - one containing all the good prefetch candidates and one containing all the

(A)

```
for (i=0; i < N; i++) {
  read(fd, buf, size);
  f(buf);
}
```

(B)

```
for (i=0; i < N; i++) {
  read(fd, buf, size);
  lseek(fd, g(buf), SEEK_SET);
}
```

Figure 5.2: Example of how the execution path taken to generate a prefetch candidate determines whether that prefetch candidate will be good or bad. In (A), if `f()` does not include any I/O calls, speculative execution of this loop would always generate good prefetch candidates. In (B), if the return value of `g()` depends on the contents of its argument, speculative execution of this loop would probably always generate bad prefetch candidates.

bad prefetch candidates. The more closely the mix of good and bad prefetch candidates in a category approaches 50-50, the less useful that category will be for the purposes of avoiding useless prefetches while issuing as many beneficial prefetches as possible. Notice that, while many likely feature sets would be fairly easy to track in a user-level design, they would often be much more complicated and expensive to track within a kernel-level design.

Deciding when to generate prefetches

There are a variety of ways to use the predicted correctness probabilities. The simplest is to set some static threshold probability and issue prefetches for candidates with predicted probabilities above that threshold. A static threshold approach is not ideal because the likelihood that, and degree to which, a useless prefetch will hurt performance depends on the abundance of memory and I/O bandwidth, which varies dynamically. Nevertheless, a simple investigation of how one might set such a static threshold is instructive.

The most important factor in determining a static threshold is whether there is any I/O concurrency. Consider the following simple performance model, which assumes that the speculating application is the only application using the I/O system and that useless prefetches never cause useful data to be ejected from memory (i.e. that there is abundant memory). Let D be the number of disks, T_{miss} be the average time to service an access to uncached data, and T_{app} be the application-specific processing time between accesses to uncached data. Assume T_{app}/leT_{miss} and that, when a normal execution stalls on an I/O, speculative execution identifies the next access to uncached data with correctness probability ρ. Also assume that, when there are multiple disks, no requests specify data that span multiple disks, and both the next uncached data accessed and the data specified by an incorrect prefetch will be equally likely to reside on any disk.

Consider a system in which all the data is stored on a single disk (i.e. $D = 1$). As shown in Figure 5.3, if the prefetch is correct, then the benefit of the prefetch will be T_{app}. If the prefetch is useless, then the cost of the prefetch will be $(T_{miss} - T_{app})$. Therefore, the

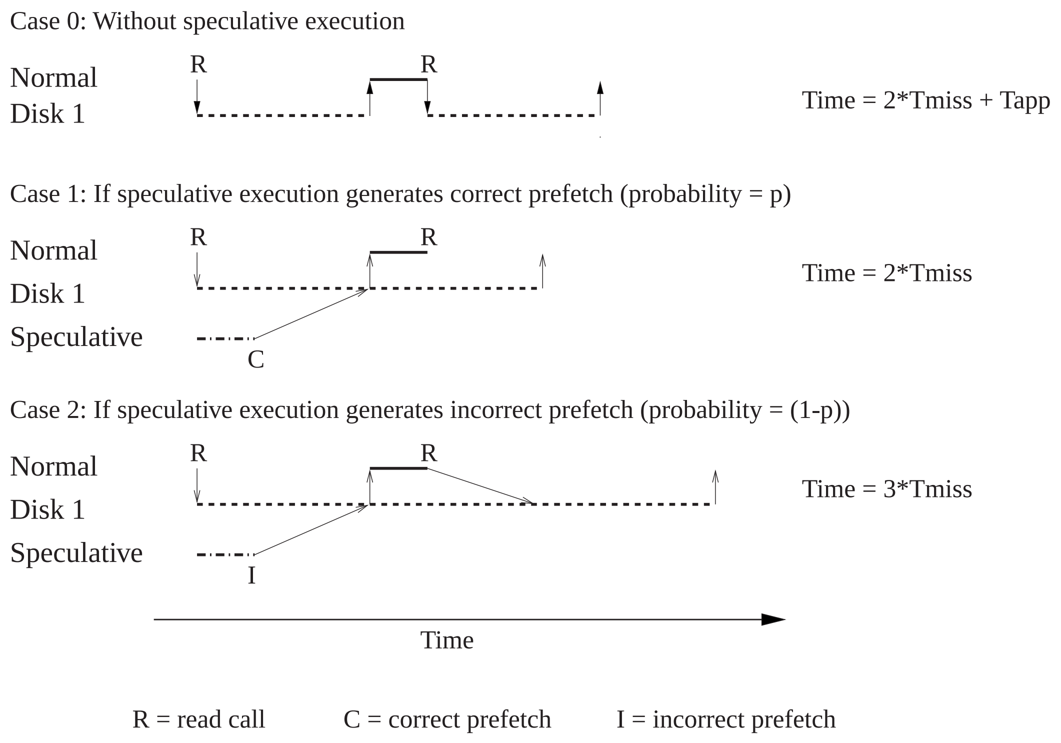
Case 0: Without speculative execution
Normal
Disk 1
R
R
Time = 2*Tmiss + Tapp
Case 1: If speculative execution generates correct prefetch (probability = p)
Normal
Disk 1
Speculative
R
R
C
Time = 2*Tmiss
Case 2: If speculative execution generates incorrect prefetch (probability = (1-p))
Normal
Disk 1
Speculative
R
R
I
Time = 3*Tmiss
Time
R = read call
C = correct prefetch
I = incorrect prefetch

Figure 5.3: How incorrect prefetches affect execution time when all data resides on a single disk, assuming there is no other competition for resources. If a prefetch should be issued whenever there is a greater probability that it will decrease than increase the execution time, then prefetches should be issued whenever ρ (the probability that the prefetch will be correct) is greater than $(1 - \frac{T_{app}}{T_{miss}})$.

prefetch should be issued only if:

$$\rho > 1 - \frac{T_{app}}{T_{miss}}$$

Therefore, since T_{app} is generally much smaller than T_{miss}, prefetches should only be issued if they are almost definitely correct.

Figure 5.4 describes the cases when there are multiple disks (i.e. $D>1$). If a prefetch should be issued whenever there is a greater probability that it will decrease than increase the execution time, then prefetches should be issued whenever:

$$\rho > \frac{T_{miss} + T_{app}(\delta D - D - \delta)}{T_{miss}(D^2 - D + 1) + T_{app}(D^2(1-\delta) + D(2\delta - 1) - \delta)}$$

where δ is the dilation factor (i.e. the relative speed of speculative execution to normal execution, which is greater than one if a speculative execution performs work that its target

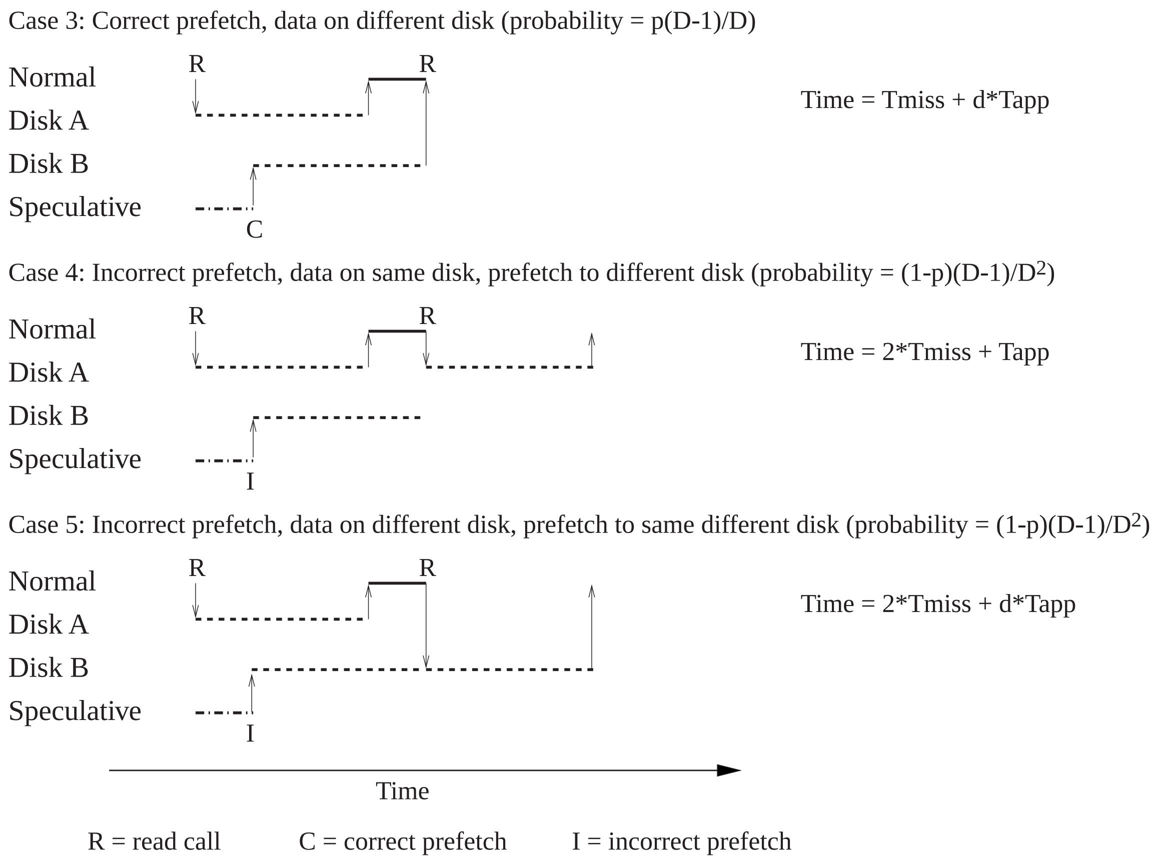
Case 3: Correct prefetch, data on different disk (probability = p(D-1)/D)
Normal
Disk A
Disk B
Speculative
R
R
C
Time = Tmiss + d*Tapp
Case 4: Incorrect prefetch, data on same disk, prefetch to different disk (probability = (1-p)(D-1)/D2)
Normal
Disk A
Disk B
Speculative
R
R
I
Time = 2*Tmiss + Tapp
Case 5: Incorrect prefetch, data on different disk, prefetch to same different disk (probability = (1-p)(D-1)/D2)
Normal
Disk A
Disk B
Speculative
R
R
I
Time = 2*Tmiss + d*Tapp
Time
R = read call
C = correct prefetch
I = incorrect prefetch

Figure 5.4: How incorrect prefetches effect execution time with multiple disks, assuming randomly distributed requests and no other competition for resources. Cases 0 1, and 2 are as illustrated in Figure 5.3, except that case 1 now occurs with probability ρ/D, and case 2 now occurs with probability $\frac{1-\rho}{D^2}$, where ρ is the probability that the prefetch is correct and D is the number of disks. The dilation factor, δ, is the relative speed of speculative execution to normal execution. Case 6 (incorrect prefetch, data on a different disk, but prefetch to the original disk) and case 7 (incorrect prefetch, data on a different disk, and prefetch to yet another disk) are not shown above. They both result in the same execution time effect as case 4 above (i.e. none). Case 6 has the same probability as case 4, but case 7 has probability $\frac{(1-\rho)(D-1)(D-2)}{D^2}$.

normal execution will not perform). Figure 5.5 shows the correctness probability for different numbers of disks if the T_{app} terms are dropped (i.e. assuming T_{app} is much smaller than T_{miss}, and δ is close to 1). Although it is likely that δ will be greater than one for a user-level design, which will increase these probabilities, this indicates that just increasing the number of disks from one to two greatly increases the tolerance for incorrect prefetches.

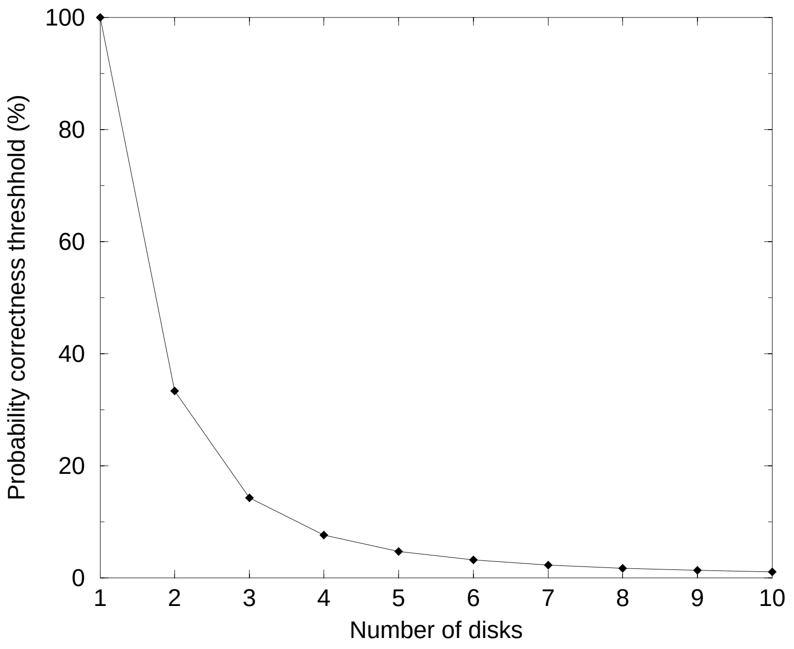

Figure 5.5: Correctness probability threshhold for varying number of disks, assuming that T_{app} is much smaller than T_{miss} and δ is close to 1, such that the T_{app} terms can be ignored. The model makes a number of assumptions (mentioned in the text). For example, the speculating application is assumed to be the only application using the I/O system, and both the data requested by the next I/O request and the data requested by an incorrect prefetch request are assumed to target each disk with equal probability.

5.1.5 Discussion

The discussions in the previous sections conclude that, on most if not all modern operating systems, a speculative execution design that is not allowed to require any operating system modifications will not be able to deliver both low overhead and high performance improvements. On the other hand, notice that operating system support sufficient to enable such a design – basically, mechanisms that would allow an execution to limit the overhead it can incur through consuming memory and I/O bandwidth – would be useful in implementing not only the speculative execution approach, but also other optimization techniques that attempt to exploit spare resources [15, 46] (including other approaches to I/O prefetching). Moreover, such mechanisms may also be directly useful to end-users. As an anecdotal example, I have found many newsgroup posts containing complaints about not being able to run a background task without hurting the performance of a foreground task due to the background task's consumption of memory and/or I/O bandwidth.

5.2 Limiting overhead in SpecHint

As discussed in the last section, a design could incur overhead by increasing the amount of work performed during normal executions, stealing processing cycles from normal executions, or as a side effect of increasing contention for main memory or I/O bandwidth. This section discusses a few elements in the SpecHint design that were shaped by the goal of low overhead. Section 5.2.1 discusses how the decision to perform speculative execution in an added thread resulted from the desire to ensure that the design neither adds a substantial amount of work to normal execution, nor steals a substantial number of processing cycles from normal execution. Section 5.2.2 discusses the limited extent to which the SpecHint design attempts to avoid incurring overhead as a side effect of increasing contention for memory or I/O bandwidth. Finally, Section 5.2.3 discusses an optional element in the SpecHint implementation for reducing useless prefetches.

The evaluation results indicate that the SpecHint design has mixed success at limiting its overhead. In particular, as shown in Table 8.7, it does not add a noticeable amount of work to normal executions, and it does not steal a noticeable amount of processing cycles from normal executions, but it can incur overhead as a side effect of increasing contention for main memory or I/O bandwidth. This last is not surprising since, as discussed in the previous section, the mechanisms provided by current operating systems do not enable a user-level design to control the performance impact of its memory and I/O bandwidth consumption without severely reducing its performance benefit.

5.2.1 The speculating thread

The design must provide a way to add speculative executions that will be able to make progress while their target normal executions are blocked. The design must also provide a way to resynchronize speculative executions with their target normal executions so that they will not waste resources running behind their target normal execution.

As discussed in Section 5.1.1, one obvious way to add a speculative execution for a target process would be to fork a child process for that target process, and then use the child process to perform speculative execution. Resynchronizing the speculative execution with its target normal execution could simply involve killing the old child process and forking a new child process. The main problem with this approach is that, on many operating systems (including Digital Unix 3.2, my evaluation system), forking is not preemptible and can take a long time, such that this approach allows speculative executions to steal noticeable amounts of processing cycles from normal executions. Therefore, the SpecHint design, does not rely on forking.

Instead, the SpecHint design takes a different approach which will not add noticeable overhead. In particular, it causes each target process to spawn a new thread when it begins executing. This new thread, which I refer to as the *speculating thread*, performs speculative execution on behalf of the process's *original thread*. The speculating thread resynchronizes with the original thread by replacing its own execution context (e.g. program counter, register values and stack) with a copy of the original thread's execution context.

The resynchronization process (which does not require any system calls) is described in detail in Section 4.2.3. The relevant point here is that spawning the speculating thread and supporting resynchronization adds only a negligible amount of work to normal execution. In addition, the SpecHint design prevents speculative executions from stealing a noticeable amount of processing cycles from normal executions by: 1) ensuring that speculating threads are assigned the lowest possible scheduling priority, and 2) preventing speculating threads from issuing any nonpreemptible system calls that could take a long time to complete. The details of how these are accomplished are described in Section 6.3.

The decision to perform speculative execution in the same address space as normal execution is the design decision with the greatest impact on the rest of the SpecHint design. In particular, as discussed further in the next chapter, it greatly complicates the mechanisms necessary to ensure that the design is safe since there are many ways in which executions in the same address space can affect one another.

One alternative approach, which may also avoid adding noticeable overhead, would be to modify normal execution such that it will replace a blocking read call for data that is not in memory with an asynchronous read call for that data, and then performs speculative execution during the interval between issuing the asynchronous read call and detecting that the data has been fetched into memory.[2] Notice, however, that such a design would also have the property that speculative execution would be performed in the same address space as normal execution. Therefore, this alternative would require similarly complicated safety mechanisms. In fact, as discussed in Section 6.2.2, it may require more complicated mechanisms. Also, it is unclear how one would extend a design based on replacing blocking read calls with asynchronous read calls to take advantage, on a multi-processor, of spare cycles while normal execution is not blocked on a read call.

5.2.2 Memory and I/O bandwidth consumption

As discussed in Section 5.1, to develop a user-level design that limits the overhead it could incur through increasing memory or I/O bandwidth consumption, it would be necessary to severely reduce the design's performance benefit. Therefore, in order to better explore the potential performance benefits of the speculative execution approach, the SpecHint design does not attempt to limit the overhead it can incur as a result of increasing memory or I/O bandwidth consumption.

The SpecHint design does include a few mechanisms for reducing this overhead. In particular, it will sometimes block a speculative execution until it can be resynchronized with its target normal execution. I refer to this as "aborting the current speculation", where a speculative execution is said to have begun a new *speculation* each time it resynchronizes with its target normal execution. One of the reasons that the SpecHint design will abort

[2]Since, on most systems, an asynchronous read is much more expensive than a synchronous read if the requested data is in memory, replacing every blocking read call (rather than just blocking read calls for data that are not in memory) with an asynchronous read call would not be an appealing approach because it could add a noticeable amount of work to a normal execution that issues many read calls for data that are in memory.

a speculation is if it predicts that the speculative execution will not be able to initiate any accurate prefetches using its current execution context, and therefore should be prevented from consuming (and wasting) resources. For example, it aborts the current speculation if a speculative execution triggers an exception which, by default, would cause the process to terminate because this probably indicates that the speculative execution has gotten completely confused through its use of incorrect data values. The mechanics of aborting a speculation are described in the next chapter.

As discussed in Section 3.2.3, the SpecHint design assumes operating system support for low overhead I/O prefetching. In particular, it assumes that the operating system supports *prefetch hint* calls that allow a user-level execution to specify precisely what file data it believes a process will access in the future. It further assumes that the operating system will schedule prefetches based on these prefetch hints in a manner that will make good use of the machine's memory and I/O bandwidth, assuming the prefetch hints are accurate. As discussed in Section 2.2.3, the purpose of such support for I/O prefetching is to enable user-level executions to initiate prefetching without needing to worry about how (by increasing contention for memory and I/O bandwidth) this prefetching could hurt performance. The SpecHint design benefits from this support in exactly this manner.

5.2.3 Filtering

The SpecHint implementation includes, as an optional element, a mechanism for identifying and avoiding useless prefetches. I refer to this mechanism as the *filtering* mechanism.

The filtering mechanism predicts prefetch correctness probabilities using a path-based approach, and then decides whether to issue prefetches by comparing these predictions against a static correctness probability threshhold. As discussed in Section 5.1.4.1, a path-based prediction approach is simply an approach that categorizes prefetch candidates based on some feature set of the execution path that was taken to generate each prefetch candidate. In the SpecHint implementation, the value used to categorize prefetch candidates – the *feature set identifier* – is a hash of the top three values on a stack of return addresses (explained further below) and the value of the stack pointer when the prefetch candidate is generated. I make no claim that this is the best (or even a particularly good) feature set identifier, either generally or for the benchmark suite I used to evaluate this implementation. Experiments (not reported on) with some other feature set identifiers do, however, indicate that this is a fairly good feature set identifier for my benchmark suite, such that the results of evaluating this implementation of filtering are interesting.

The implementation of the filtering mechanism does not add any work to normal execution. It adds two data structures to the binary: a stack of return addresses, and a data structure which maintains two counters per unique feature set identifier that is being tracked by the speculating thread. A feature set identifier starts being tracked the first time it is detected that a prefetch with that identifier was useless. Tracking only feature set identifiers that have been associated with a useless prefetch decreases the run-time cost of the implementation. The two counters for each feature set identifier basically indicate the number of prefetches issued that had this identifier, and the number of those prefetches that were subsequently detected to have been useless.

The stack of return addresses is maintained by the speculating thread, which pushes the appropriate return address value onto the stack when it makes a procedure call, and pops a value off the stack when it returns from a procedure call.[3] Upon generating a prefetch, the speculating thread records the prefetch's feature set identifier in an in-memory log and, if the identifier is being tracked, increments the associated count of prefetches issued with that identifier. When a resynchronization is triggered in response to detecting a useless prefetch (as discussed in Section 4.2.2), the speculating thread increments the appropriate feature set identifier's counter of useless prefetches. (It also decrements the count of issued prefetches for all subsequent, tracked identifiers in the log.) Finally, before generating a prefetch with some identifier, the speculating thread checks whether the percentage of issued prefetches with that identifier that were detected to have been useless is below some static threshhold. If so, it issues the prefetch; otherwise, it issues the prefetch with some low probability in order to enable more information to be gathered about any feature set identifier. If the speculating thread does not issue the prefetch, then it aborts the current speculation under the conservative assumption that the current speculation has gotten confused by incorrect data values, and therefore should avoid consuming any more machine resources.

5.3 Summary

A design for adding speculative executions could incur overhead by increasing the amount of work performed during normal executions, stealing processing cycles from normal executions, or as a side effect of increasing contention for main memory or I/O bandwidth. It is fairly straightforward to avoid incurring noticeable overhead in either of the former two ways. On the other hand, there is no easy way to guarantee that a design will not incur noticeable overhead in the latter way. Moreover, a design that is not allowed to require operating system modifications will need to chose between sometimes incurring high overhead or severely reducing its performance benefit.

Since it is difficult to avoid incurring overhead as a side effect of increasing contention for main memory or I/O bandwidth, a design might benefit from attempting to reduce the useless I/Os that will be initiated by speculative executions. On the other hand, a design can unnecessarily reduce its performance benefit by not issuing prefetches that would have been correct. There are a variety of ways in which a design could attempt to predict the probability that a prefetch will be correct. Back-of-the-envelope calculations indicate that, to optimize predicted performance, only prefetches that are very likely to be correct should be issued on a system with a single disk. On systems with more than one disk, however, the ability to service multiple requests in parallel can more easily hide the cost of useless prefetches, such that it may make sense to issue prefetches that are less likely to be correct.

[3]Actually, to limit the amount of extra work added by this technique, no values are added/removed from the stack of return addresses for calls which could not lead to a read call, or for calls from procedures below a certain size.

The SpecHint design will not incur noticeable overhead through increasing the amount of work performed during normal executions, or stealing processing cycles from normal execution. However, it may incur noticeable overhead as a side effect of increasing contention for memory and I/O bandwidth. The SpecHint design relies on operating system support for low overhead I/O prefetching.

Overhead considerations shaped the single most defining design choice in the SpecHint design. In particular, speculative execution in the SpecHint design is performed in the same address space as normal execution (by an added speculating thread). To reduce its resource wastage, the SpecHint design will abort the current speculation when it seems unlikely to generate additional accurate prefetches. Finally, as an optional element, my SpecHint implementation supports filtering; that is, it can predict correctness probabilities for prefetch candidates, and use these predictions to decide whether or not to issue a prefetch.

Chapter 6

Design goal: Safety

A design for adding speculative executions will cause the behavior of a system to change in a variety of ways. After all, to be effective, speculative execution must change the system's behavior; in particular, it must reduce the I/O stall time of target normal executions by fetching data earlier than it would otherwise be fetched. However, a design should strive to avoid causing changes that are "unsafe", where a change is unsafe if users would consider that change, or any consequence of that change, to be a malfunction. This is an important design goal because, even if the design can deliver huge performance benefits, it is much less likely to be adopted if users believe that it causes their systems to malfunction.

It may be possible to construct pathological circumstances in which any change in the behavior of a system might cause what a user would perceive as a malfunction. Therefore, any design will be based on some assumptions about what changes are unsafe. These assumptions determine the *safety* of the design. If, in practice, these assumptions are rarely if ever false for some set of applications, then the design might be considered safe for that set of applications.

In order to evaluate the safety of a design, it is necessary to understand the assumptions on which it relies, and the circumstances in which those assumption might prove false. Section 6.1 discusses an assumption used throughout this chapter. Then, Section 6.2 discusses how different designs might limit the additional assumptions on which they rely by limiting the ways in which they could possibly change system behavior. Finally, Section 6.3 describes the set of assumptions on which the SpecHint design is based, and how it guarantees that this set is complete.

6.1 Base safety assumption

An execution changes the behavior of the system by producing *direct output* ("user output", e.g. what it causes to be displayed on a screen and any changes it makes to the contents of file systems) and *indirect output* (any additional ways it can affect other executions, e.g. its consumption of shared resources and any messages it sends to other executions). In turn, a system's behavior can be thought of as the combined output of all executions on that system. For the purposes of this dissertation, to avoid over-constraining the design space, I make the following simplying assumption about what output could cause unsafe changes in system behavior:

Base assumption. I assume that users will not consider any changes in system behavior that result from changes in shared resource usage to be malfunctions, where I define *shared resource usage* to be the allotment of processing resources, physical memory, I/O bandwidth, and operating system resources (i.e. data structures maintained by the operating system).

This assumption may not always be true in practice. For example, while most operating systems support the illusion of having an infinite amount of processing resources and I/O bandwidth, operating systems do not support the illusion of having an infinite amount of physical memory or operating system resources. In particular, most operating systems limit the amount of memory and/or different operating system resources that can be allocated system-wide, per user, and/or per process, and reject allocation requests for resources whose limits have been reached. For example, most operating systems limit the total and per-user number of processes, and will reject process creation requests while at that limit. Moreover, many programs are implemented such that, if the operating system rejects an allocation request by a normal execution of the program, the normal execution would behave in a manner that a user would probably consider a malfunction. For example, many programs are implemented such that, if the operating system rejects a request for more memory, execution will abort with an error message. Therefore, allowing speculative execution to consume memory or operating system resources could result in the operating system rejecting a subsequent allocation request by a normal execution, which could result in an unsafe change in the behavior (i.e. the output) of that normal execution. Fortunately, the limits on memory and operating system resources are usually high enough that they are seldom reached in practice unless there is a “resource leak” (i.e. unless resources are allocated and never released). Therefore, I assume that changes in shared resource usage cannot cause malfunctions, unless they introduce a leak of memory or operating system resources.

Notice that changes in shared resource usage will often result in changes in timing and/or resource usage information maintained by the operating system. For example, changing which data is in physical memory could change how long it takes for an execution to access its data, and how many I/O requests are performed on behalf of that execution. This suggests two more notable circumstances in which the above assumption may fall short. First, the assumption may fall short for real-time applications. In particular, a change in timing might cause a normal execution of a real-time application to miss a deadline that it otherwise would not have missed. A user might consider such a missed deadline to be a malfunction. Second, it may fall short for applications that contain race conditions that are exposed by adding speculative execution. While such race conditions could be considered defects in the original application, users may blame speculative execution for affecting system behavior in such a way that they become manifest.

To simplify the text in the remainder of this chapter, the above assumption is assumed to be true, so long as no resource leaks are introduced. Thus, for example, when I refer to the indirect output, or the output, of an execution, I am excluding the execution's shared resource usage, or any effect thereof.

6.2 Designing for safety

Since an execution can only affect the system's behavior by producing output (where output includes both direct and indirect output, as defined in the previous section), a design would provide complete safety if it could guarantee that speculative execution (and any added normal execution) could not produce output. Therefore, one strategy for developing a design would be to identify every action that speculative or added normal execution might perform that could allow the speculative or added normal execution to produce output, and then attempt to guarantee that the design prevents speculative and added normal execution from performing any such actions.

For example, on most UNIX operating systems, an execution can produce *direct* output only by performing specific system calls (for example, system calls that might change the contents of a file system), modifying a mapped file, or accessing a mapped device [64]. Therefore, a design could ensure that speculative execution (or added normal execution) could not produce direct output by guaranteeing that the execution could not perform any system call that might produce direct output, could not modify any mapped files, and could not access any mapped devices.

On the other hand, the set of actions that might allow an execution to produce *indirect* output will depend heavily on whether the execution: 1) does not share an address space with any other execution, 2) shares an address space but not a thread with some other execution, or 3) shares a thread with some other execution. Sections 6.2.1, 6.2.2, and 6.2.3 begin by identifying the actions that could produce indirect output in each of these cases, respectively. In the first case, the section proposes a set of guarantees that would be sufficient for a design to ensure that such an execution could not produce (direct or indirect) output. In the latter two cases, it may not be practical to prohibit one or more of the actions that could produce indirect output. On the other hand, it may be possible ensure that such an action could cause unsafe changes only in unusual circumstances. Therefore, these sections instead propose sets of guarantees that would be sufficient for a design to ensure that such an execution could produce unsafe changes only in unusual circumstances, as described in additional assumptions.

The rest of this section assumes a UNIX operating system (e.g. "most operating systems" means "most UNIX operating systems").

6.2.1 Separate process

Consider a design in which a speculative execution (or added normal execution) takes place in an address space that is not shared with any other execution, e.g. speculative execution in a design based on forking like those sketched in Section 3.2.2. I will refer to such an execution as an *added process*. Such a design can exploit existing hardware- and operating system-enforced protection boundaries between processes to ensure that the added process cannot produce indirect output. In particular, on a well-implemented system, these protection boundaries ensure that processes can produce indirect output only by terminating, performing specific system calls (for example, system calls that allow the process to communicate with another process), modifying shared memory segments or mapped file, or accessing mapped devices [64].

An added process could produce indirect output by terminating because, on most operating systems, terminating a process causes a child termination signal ($SIGCHLD$) to be delivered to the process's current parent, and may also result in the process's exit status being delivered to its parent. An added process can terminate by either issuing a system call that causes it to terminate, or receiving a signal that causes it to terminate. While a design could prevent the former by prohibiting added processes from issuing such system calls, it is not possible to prevent the latter on most systems. Moreover, preventing added processes from terminating would introduce a resource leak. Therefore, a design should ensure that added processes will eventually terminate and be reclaimed (releasing all the resources they consume). If a design is allowed to include operating system modifications, it could ensure that added processes can terminate and be reclaimed, without producing indirect output, by requiring that the operating system's process termination routine be modified. In particular, it could require that the routine be modified such that, when an added process terminates, its exit status is discarded and its resources reclaimed, without sending a signal to its parent.

A design could prohibit an added process from modifying any shared memory segments or mapped files by ensuring that any memory segments and files mapped into the added process's address space are mapped as read-only or private memory (where memory is private if the system guarantees that any modification of that memory cannot produce indirect output). It could prohibit an added process from accessing any mapped devices by ensuring that devices are never mapped into the added process's address space. (Notice that, if an added process may be created with mapped shared memory segments, files and devices (which is the case when forking a process on most operating systems), it would not be sufficient to simply limit the system calls that added processes could issue to map shared memory segments, files or devices.)

In summary, a design could guarantee that an added process (whether used for speculative or added normal execution) could not produce output by ensuring that:

- The added process will terminate and have its resources reclaimed without producing output.
- The added process is prohibited from performing any system calls that could allow it to produce output.
- Shared memory and files are mapped into the added process's address space only as read-only or private memory. And,
- Devices are never mapped into the added process's address space.

6.2.2 Shared process, separate thread

Now, consider a design in which a speculative execution (or added normal execution) shares an address space with another execution, but uses a separate thread, e.g. speculative execution in the SpecHint design (as discussed in Section 3.2.3). I will refer to such an execution as an *added thread*, as opposed to the *original thread(s)*. The threads in a particular design may be either kernel threads or user threads; that is, either the operating system or a user-level threads package may be responsible for associating each thread with its own *thread state* (program counter, register values, stack, running state, plus any other per-thread state maintained by the particular operating system or user-level threads package). Threads in the same address space share *process state* (process identifier, address space, open file information, and any other per-process state maintained by the particular operating system). A thread can produce indirect output that may affect the output of another process (or of an original thread) in all the ways described in the previous section. In addition, a thread can produce indirect output that may affect the output of an original thread by modifying their shared process state, or the thread state of that original thread. On most operating systems, a thread could do this only by performing the specific system calls that could modify its process state or the thread state of another thread in the same process, modifying arbitrary memory values, triggering an exception, or receiving asynchronous signals [64].

It may not be practical for a design to guarantee that an added thread could not produce indirect output. For example, if the added thread is used to perform speculative execution, preventing the added thread from modifying any memory values would prevent speculative execution from retaining the results of computations performed while running ahead of its target normal execution. This would greatly increase the number of incorrect values used during speculative execution and, as demonstrated in Section 8.2.1, severely diminish the set of applications for which the design could provide benefit. However, it may be possible for a design to restrict the ways in which an added thread could produce indirect output in such a manner that any indirect output the added thread produced would change the output of an original thread only in unusual circumstances.

For example, most modern operating systems divide each process's address space into equal-sized units refered to as pages, and allow a process (executing at user-level) to access only pages that contain the process's code, global data, stack(s), or heap(s). If a process attempts to access any other page, it will trigger an exception, which will usually cause the operating system to terminate the process immediately. Therefore, I claim that the following assumption will be true for a broad range of applications:

Assumption 1. Original normal executions will access only pages that contain original program code, original global data, their stack(s), or their heap(s).

Based on this assumption, a design can guarantee that added threads will not, by modifying memory values, change the output of original threads by ensuring that added threads will never modify values on such pages, which I will refer to as "pages mapped for original execution".

Notice, however, that an added thread can produce indirect output by mapping other pages (that it could then modify). In particular, it can interfere with the growth of memory regions that the operating system requires to be contiguous, where, on most operating systems, the only such memory regions are the primary thread's stack and the brk heap (the heap whose bound is determined by the address in the brk pointer maintained by the operating system). It can also introduce a resource leak by progressively mapping an increasing number of pages. Finally, it can change which pages the operating system maps in response to subsequent mapping requests by original threads. This last case, however, may not be an issue. On most systems, if an execution demands particular pages (as it can do via $mmap$ or $shmat$ system calls, for mapping files, devices or memory, and attaching memory segments, respectively), the operating system will grant those pages even if they were already mapped (unmapping whatever used to be in those pages). Therefore, an added thread's page mappings could change only which pages the operating system will map in response to mapping requests that claimed not to require particular pages. Thus, I claim that the following assumption will be true for a broad range of applications:

Assumption 2. Unsafe changes will not result from changing which pages an operating system will map in response to mapping requests from original normal executions (that do not specify particular pages).

Based on this assumption, a design can guarantee that added threads will not, by mapping pages, cause unsafe changes in the output of original threads by ensuring that such mappings could not cause a resource allocation request by an original thread to fail (either by causing a resource limit to be met, or by interfering with the growth of the primary stack or brk heap).

Next, consider how allowing an added thread to trigger an exception could produce indirect output. Whenever a thread triggers an exception, the operating system will generate a signal for that thread of a particular type (depending on how the thread triggered the exception). I will refer to the types of signals that could be generated by triggering an exception as the *exception signal types*. What happens after a signal is generated depends on the action specified for that type of signal, and whether that type of signal is "blocked". On most operating systems, the action specified for each exception signal type can be one of the following: 1) ignore the signal, 2) terminate the process (the default), or 3) execute the appropriate *signal handler* (the function previously designated as the function to execute upon receiving a signal of that type). If the action specified for that type of signal is to ignore the signal, then the operating system will discard the signal. Otherwise, the operating system will deliver the signal as soon as that type of signal is not blocked (e.g. immediately, if that type of signal is not blocked when the signal is generated); that is, depending on the action specified, the operating system will either terminate the process, or cause the triggering thread to execute the designated signal handler. Therefore, by triggering an exception, an added thread could produce indirect output by causing the process to be terminated, or by executing a signal handler that allows it to produce indirect output.

Notice that, unless it is prevented from doing so, it is not only possible but also quite likely that an added thread used for speculative execution will trigger exceptions because such a thread will sometimes uses incorrect data values (as discussed in Section 3.1.2). In particular, it is not unlikely that an added thread used for speculative execution will, as a result of using incorrect data values, trigger an arithmetic exception, or calculate memory addresses incorrectly such that it triggers an exception by attempting to access an invalid address or memory that is inaccessible.

One way for a design to guarantee that an added thread could not produce indirect output by triggering an exception would be for it to guarantee that added threads could not trigger exceptions (i.e. by statically and/or dynamically ensuring the validity of all operations that could potentially trigger an exception). However, this may not be practical when considering all the ways a thread could trigger an exception. An alternative would be for a design to guarantee that, for all the types of signals that it allows added threads to generate by triggering exceptions, the specified action is neither to terminate the process, nor to execute a signal handler that might allow the thread to produce indirect output. This is complicated by the fact that, on most operating systems, the action specified for each type of signal is shared by all threads in a process, so changing the actions specified for an added thread could produce indirect output by changing the actions specified for original threads. If a design is allowed to include operating system modifications, then it could require that the operating system be modified to allow added threads to have different actions specified. Otherwise, it could approximate this effect at user-level by specifying signal handlers that would cause different code to be executed depending on whether they were being executed by an added or original thread. For example, if the specified action for original threads would be to execute a particular function, then a design could specify a signal handler that would cause any original thread to jump to the entry address of that function (while causing any added thread to perform some other computation).

However, there is one case in which such a user-level design may not be able to guarantee the same effect. In particular, an operating system requires some amount of stack space to deliver a signal for which the specified action is to execute a signal handler, even if the signal handler itself requires no stack space to execute. If the operating system attempts to deliver such a signal when the stack that would be used to handle the signal has insufficient space, then an exception will be triggered. Therefore, simply by changing the specified action for some type of signal from ignoring the signal or terminating the process to executing some signal handler, a design could cause an extra signal to be generated for an original thread. If the specified action for this extra signal is anything other than ignoring the signal, then the process will be terminated. Assume that a design causes the specified action for this signal to be executing a signal handler. If the specified action would have been to ignore the signal, the design could cause the process to terminate prematurely. Notice, however, that, if a signal generated by triggering an exception is ignored, most systems will simply re-execute the instruction that triggered the exception, which would cause the exception to be re-triggered repeatedly. Therefore, I claim that the following assumption will be true for a broad range of applications:

Assumption 3. The following will not both occur: 1) an original normal execution receives an exception signal while it has insufficient stack space to handle a signal, and 2) it would have specified that the action for that exception signal type be to ignore the signal.

On the other hand, if the specified action would have been to terminate the process, then the design could cause the process to terminate with a different exit status. However, this could only cause an unsafe change if the process's parent actually examines its exit status, and is affected by exactly which exception signal caused its child to terminate. Therefore, I claim that the following assumption will be true for a broad range of applications:

Assumption 4. The following will not both occur: 1) an original normal execution receives an exception signal while it has insufficient stack space to handle a signal, and 2) a change in its exit status will cause an unsafe change in its parent's output.

Based on these assumptions, a design can guarantee that added threads will not, by triggering exceptions, cause unsafe changes in the output of original threads by ensuring that it will handle exception signals in a manner that does not produce any output, and does not change the effect of exception signals generated for original threads except in the boundary cases described above.

Finally, consider how allowing an added thread to receive asynchronous signals could produce indirect output. Most operating systems will deliver an asynchronous signal of a particular type to any thread that is not blocking signals of that type. (Most modern operating systems support per-thread signal masks, where a process/thread's *signal mask* specifies which types of signals are blocked for that process/thread.) Therefore, if an added thread is not blocking some type of signal, then it could produce indirect output by receiving an asynchronous signal of that type, that would otherwise have been delivered to another thread. A design could guarantee that it will not allow an added thread to, by receiving an asynchronous signal, produce indirect output by guaranteeing that added threads always block all types of signals. However, consider a design that allows added threads to generate exceptions. If the design is allowed to include operating system modifications, then it could guarantee that added threads could not, by receiving asynchronous signals, produce indirect output by requiring that the operating system be modified such that it will never deliver asynchronous signals to added threads. On the other hand, if the design is not allowed to include operating system modifications, then it would be able to guarantee that added threads could not produce indirect output only by guaranteeing that, for all exception signal types associated with exceptions that added threads could trigger, the action specified is either to ignore the signal or to execute a signal handler that would not allow added threads to produce indirect output. The former would not be a good design decision for the reasons discussed in the previous paragraph, and the latter requires that added threads not block the appropriate exception signal types whenever they might trigger an exception. I will refer to signals of a type that could be generated by triggering an exception as *exception signals*. Most operating systems will generate an exception signal as an asynchronous signal only in response to an explicit signal generation request (e.g. a $kill$ system call). In addition, most operating systems supply other signal types that are expressly for use in programs that wish to communicate by explicitly generating signals. Therefore, I claim that the following assumption will be true for a broad range of applications:

Assumption 5. Exception signals are not used as asynchronous signals.

Based on this assumption, a design can guarantee that added threads will not, by receiving asynchronous signals, cause unsafe changes in the output of original threads by ensuring that they always block all but exception signals.

In summary, it would be very difficult for the design to guarantee that an added thread could not possibly produce indirect output. However, it is possible to limit the ways in which it could produce indirect output. In particular, based on the assumptions detailed in this section, I propose that a design could guarantee that an added thread (whether used for speculative or added normal execution) would cause no unsafe changes for a broad range of applications by ensuring that:

- The added thread can map only pages not mapped for original execution.
- Any pages mapped by added threads are unmapped before their being mapped causes the operating system to reject a mapping request by an original thread (either by causing a resource limit to be met, or by interfering with the growth of the primary stack or brk heap).
- The added thread is prohibited from mapping shared memory segments, files, or devices into its address space.
- The added thread is prohibited from performing any system call that could allow it to produce output (where the above is assumed to be sufficient for ensuring that mapping pages cannot produce output).
- The added thread does not modify any pages mapped for original execution.
- Either the added thread cannot trigger exceptions, or any exceptions that could be triggered by the added thread cannot change its output. And,
- The added thread blocks all signals except exception signals.

6.2.3 Shared thread

Finally, consider a design in which a speculative execution or added normal execution shares a thread with another execution. Such an execution could produce indirect output in the ways that a separate thread could produce indirect output (as described in the previous section). In addition, it could produce indirect output by changing what the thread's state would be during the other execution. On most operating systems, a thread can modify its thread state only by triggering an exception, receiving a signal, performing a system call that could modify its own thread state, or modifying its program counter, register values or stack [64].

A design in which speculative execution shares a thread with another execution can be converted into one in which speculative execution shares an address space but not a thread with another execution. This can be achieved by treating speculative execution and the execution with which it shares a thread as two virtual threads multiplexed onto that actual thread, e.g. by logically adding another layer of thread support.

On the other hand, to add work to normal execution, it may be more natural for a design to simply insert short sequences of instructions into the sequence of instructions executed during original normal execution. I will refer to these inserted sequences as *side trips*. A design could guarantee that added normal execution will not produce indirect output as the result of either triggering an exception or receiving a signal, by guaranteeing that side trips could not trigger exceptions and that all signals will be blocked during side trips. Also, a design could guarantee that added normal execution will not produce indirect output as the result of changing register or stack values by ensuring that, upon ending each side trip, the only register and stack values the side trip changed were *dead values* (values that would not have been used before they were changed anyhow).

Therefore, based on the assumptions in the previous section, I propose that a design could guarantee that added normal execution, sharing a thread with its original normal execution, would cause no unsafe changes for a broad range of applications by ensuring that: 1) any pages mapped during added normal execution are unmapped before their being mapped causes the operating system to reject a mapping request by original normal execution, and 2) added normal execution consists of side trips, where each side trip:

- Begins by changing the thread's signal mask to block all signals and end by changing the thread's signal mask back to its prior setting.
- Can map only pages not mapped for original execution.
- Except for the above, cannot perform any system call which might produce output.
- Cannot map shared memory segments, files, or devices into the address space.
- Cannot modify any pages mapped for original execution, except memory that contains the thread's stack.
- Cannot trigger any exceptions.
- Must complete. And,
- Upon completion, must leave the thread state (including the thread's stack and signal mask) as it was when the side trip began, except that it need not restore dead values, and that the program counter should now indicate what would have been the next instruction had the side trip not occurred.

6.3 Safety of SpecHint

In the previous section, I proposed sets of guarantees that I claim a design could provide to ensure that it would cause no unsafe changes for a broad range of applications. In this section, I instead focus on the issues that arise in trying to provide such guarantees in the context of a particular design, the SpecHint design.

The SpecHint design is based on binary modification. It specifies the behavior of a binary modification tool and the behavior of a set of support routines that the binary modification tool requires in addition to the application binary. Throughout this section, when I refer to "what the SpecHint design guarantees", I mean what an implementation of the SpecHint design would guarantee if it met the specification and introduced no safety loopholes as a result of how it implemented the design. The SpecHint implementation refers to my implementation of this design (tool and support routines), while the *SpecHint tool* refers to just my binary modification tool.

The SpecHint design is targeted at single-threaded applications. It is structured such that speculative execution will be performed in the same address space as its target normal execution, but by a separate, added thread. I refer to this thread as the *speculating thread*. Added normal execution, on the other hand, will share a thread with original normal execution. I will refer to this thread as the *original thread*. The support routines specified by the design will be included in every transformed binary. These routines provide functionality essential to the design, and not specific to any application. They can be divided into support routines for speculating threads, support routines for original threads, and the *exception handling support routine* (for both threads).

The design limits what code speculating threads could possibly execute. In particular, the design guarantees that speculating threads can execute only support routines for speculating threads, the exception handling support routine, a few system calls which will produce no output (which will be the case if they behave as commonly expected), and a specific subset of the code in each transformed application binary that I refer to as *shadow code*. While the support routines are common to all applications, shadow code is the application-specific code that speculating threads execute to run ahead of their original threads. The design specifies what a binary modification tool should accomplish to generate shadow code automatically in a manner that, combined with the design's specifications for the support routines, guarantees that speculating threads and added normal execution would cause no unsafe changes for a broad range of applications.

The SpecHint design is based on the assumptions detailed in Section 6.2.2; that is, throughout this section, whenever I refer to "indirect output", I am excluding indirect output that, as discussed in that section, would not cause unsafe changes for a broad range of applications. Also, whenever I refer to executing some code in the binary, I am including performing any system calls issued while executing that code.

This section begins by describing what shadow code is, key properties of shadow code, the specifications in the SpecHint design that guarantee those properties, relevant implementation choices in the SpecHint implementation, and tradeoffs in basing a design on shadow code. It then describes the specified behavior of the support routines, when those routines should be called, and how these specifications combine to establish the safety of the design.

6.3.1 Shadow code

Shadow code is the application-specific code that speculating threads execute to run ahead of their original threads. In particular, for a given application, it is an extensively modified copy of the code from the original application binary, that the SpecHint design specifies should be generated by the binary modification tool and included in the application's transformed binary.

I use the term *shadow memory* to refer to the memory allocated to hold a speculating thread's stack, plus any memory allocated via one of the *shadow allocation support routines* (a subset of the support routines for speculating threads, described in Section 6.3.2.2). The SpecHint design specifies how to guarantee that, while executing shadow code, speculating threads:

1. Will not be able to modify any memory other than shadow memory.
2. Will not be able to issue any system calls other than those which are guaranteed to return to user-level code, and can change at most the values in their registers and the contents of one or more buffers entirely contained within shadow memory.
3. Will always block all but exception signals, and will always have enough space in the stack(s) that would be used for signal handling to handle any signals they might receive. And,
4. Will be able to "escape" from shadow code to only the exception handling support routine, or one of the support routines for speculating threads.

Notice that, if shadow memory does not overlap any pages allocated for original execution, and if the actions specified for exception signals will not allow speculating threads to produce output, then the first three guarantees are sufficient to ensure that, while executing shadow code, speculating threads would not produce output (for the reasons discussed in Section 6.2.2). The last makes it possible for the SpecHint design to restrict what code speculating threads could execute (and therefore what unsafe actions they could perform) simply by specifying the behavior of those support routines appropriately.

To generate shadow code, the SpecHint design specifies that the binary modification tool make a copy of the code section in the original application binary, and then modify this copy in a certain manner. In particular, the SpecHint design specifies several types of *checks* and *check support routines*. A check is a sequence of instructions that may include a call to a check support routine, the check support routines are a subset of the support routines for speculating threads, and each check – in combination with whatever check support routine it calls – enforces a particular constraint. The design also specifies when a speculating thread must perform these checks while executing shadow code.

Typically, each check is associated with some instruction, which I refer to as the *checked instruction*, and is responsible for guaranteeing that this instruction is only executed if doing so would not violate some specification. If executing the instruction would violate the specification, then the check (or the check support routine it calls) would ensure that it is not executed by aborting the current speculation; that is, it will cause the speculating thread to resynchronize by calling a particular support routine for speculating threads, the *resynchronization support routine*. The SpecHint design specifies that this routine allow the speculating thread to leave the routine only by jumping to particular addresses in shadow code (and only after updating its state in a manner described in the next chapter); that is, from the perspective of any function that calls this routine, the routine never returns.

Sections 6.3.1.1, 6.3.1.2, 6.3.1.3, and 6.3.1.4, begin by describing the specifications for the check(s) and check support routine(s), and any other relevant specifications, with which the SpecHint design provides each of the above guarantees, respectively. Each section ends by describing relevant implementation decisions in the SpecHint implementation. To simplify the text in this section, although the copy of the code is not technically "shadow code" until all the necessary modifications have been made, I will refer to it as shadow code throughout this section.

6.3.1.1 Software copy-on-write

This section discusses how the design guarantees that, while executing shadow code, speculating threads will modify only shadow memory (the memory allocated to hold a speculating thread's stack, plus any memory allocated via one of the shadow allocation support routines). Section 6.3.2.2 will discuss how the design ensures that shadow memory does not overlap pages allocated for original execution, such that this guarantee is sufficient to ensure that, while executing shadow code, speculating threads cannot change the output of normal execution by modifying memory values that are subsequently used during original normal execution. The section begins by describing a well-known technique that could be leveraged to provide this guarantee. It then describes the mechanism by which most systems support this technique, and why that mechanism is inappropriate for the SpecHint design. Next, it discusses the alternative mechanism specified by the SpecHint design. Finally, it discusses a few decisions in the SpecHint implementation.

Copy-on-write is a technique for allowing two (or more) entities to share some data, while ensuring that, if either of them modifies the data, the data seen by the other entity will not be changed. The data may be divided into one or more units, and each unit is initially designated a copy-on-write unit. Copy-on-write involves detecting the first attempt to modify a copy-on-write unit by either entity and, at that time, making a copy of that unit, and ensuring that each of the entities will subsequently access its own (no longer copy-on-write) version of that unit.

The technique is called copy-on-write because a copy is made only upon the first attempt to modify a copy-on-write unit. Notice that an alternative approach would be to copy all the data upfront, rather than allowing the data to be shared. This alternative approach has the advantages of not requiring a mechanism for detecting the first attempt to modify a copy-on-write unit, and not incurring any runtime cost associated with detecting these modification attempts. However, copy-on-write is often more efficient for two reasons. First, copying all the data upfront could take a considerable amount of time. Second, if neither entity attempts to modify some copy-on-write unit, then that unit would never need to be copied, saving both time and space.

Notice that copy-on-write could be leveraged to provide the desired guarantee that, while executing shadow code, speculating threads will be able to modify only shadow memory. In particular, for each execution of a transformed application binary, the shared data would be *non-shadow memory* (i.e. all memory that is not shadow memory), and the entities would be the original thread and the speculating thread while it is executing shadow code. Non-shadow memory would be divided into one or more units, all initially designated copy-on-write. Then, assuming a suitable copy-on-write mechanism, a design could guarantee that the speculating thread would be able to modify only shadow memory simply by ensuring that, regardless of which thread initiates the first attempt to modify a copy-on-write unit, the original thread would always retain the original copy of the unit.

Most systems provide a copy-on-write mechanism that I will refer to as *standard copy-on-write*. In particular, most systems divide each process's address space into equal-sized units called pages, and support copy-on-write between processes at the granularity of a page. On most systems, the first attempt to modify a copy-on-write page is detected in hardware, in the process of address translation. Upon detecting that a copy-on-write page needs to be copied, the hardware generates an exception, which causes the operating system to execute. The operating system manages the actual copying of copy-on-write pages and the updating of address spaces to ensure that each process will subsequently access its own (no longer copy-on-write) page.

Since detecting when a page should be copied is performed in hardware, the the copy-on-write provided by most systems is very efficient. Unfortunately, it is inappropriate for the SpecHint design because operating systems support copy-on-write only between processes, not between threads in the same process.

The SpecHint design supports a variant of “ordinary” copy-on-write which I refer to as *selective copy-on-write*. Recall that, as described above, copy-on-write involves making a copy upon the first attempt to modify some data shared by multiple entities, regardless of which entity attempts to modify the data. Selective copy-on-write is functionally identical to copy-on-write except that it makes a copy only upon the first attempt to modify the shared data by an entity that is a member of a pre-specified subset of the entities sharing the data. That is, the shared data can be modified by any entity not in that subset without causing the data to be copied.

The SpecHint design specifies a mechanism, which I refer to as *software copy-on-write*, for implementing, entirely at user-level, selective copy-on-write between threads in the same address space. The SpecHint design guarantees that speculating threads can modify only shadow memory while executing shadow code by specifying the behavior of two types of checks (collectively refered to as *copy-on-write checks*), the behavior of the check support routines they call (the *store check support routine* and the *load check support routine*), and what the binary modification tool must ensure in modifying shadow code to include copy-on-write checks.

In particular, the design specifies that a process's address space be logically divided into *regions*, where each region is part of either shadow memory or non-shadow memory. The design specifies that the store check support routine behave as follows:

- It should expect an address as its argument.
- If the argument refers to a shadow memory region, then it should return the argument.
- If the argument refers to a non-shadow memory region that has a copy, then it should return the appropriate address in that copy. And,
- If the argument refers to a non-shadow memory region that does not have a copy, then it should either 1) copy the region into some memory that is shadow memory but not holding the speculating thread's stack (possibly first allocating this memory by calling one of the shadow allocation support routines for speculating threads), update whatever data structure is maintained to map between non-shadow memory regions and their copies, and return the appropriate address in the newly created copy,

or 2) abort the current speculation.

The design specifies that the load check support routine behave as follows:

- It should expect an address as its argument.
- If the argument refers to a shadow memory region, or a non-shadow memory region that does not have a copy, then it should return the argument. And,
- If the argument refers to a non-shadow memory region that has a copy, then it should return the appropriate address in that copy.

(The design further specifies that the behavior of both of these support routines be restricted as described in Section 6.3.2.3.) In addition, the design specifies that the binary modification tool modify shadow code to include sufficient copy-on-write checks. In particular, the design specifies that the binary modification tool modify shadow code such that a thread which executes shadow code will call (with the correct return address) the store/load check support routine before every store/load instruction that might otherwise store/load to non-shadow memory, specifying what would be the target address of the store/load instruction as the argument to the routine, and will then store to/load from the address specified as the routine's return value instead of the address it specified as the argument to the call. Notice that a binary modification tool could meet this specification by modifying shadow code such that a thread which executes shadow code will perform a copy-on-write check before every store and load instruction in shadow code. However, it is possible for a tool to add much less work to speculative execution. In particular, if a tool guarantees that some subset of store and load instructions in shadow code could not possibly store or load to non-shadow memory, then the tool would not need to ensure that a thread would execute a copy-on-write check before those instructions.

Through this specification, the SpecHint design guarantees that, while a speculating thread is executing shadow code, whenever it would otherwise store to a non-shadow memory region that has no copy, it will instead make a copy of that region and redirect the store to the copy (or it will abort the current speculation). Whenever it would otherwise store to/load from a copied non-shadow memory region, it will instead redirect the store/load to the copy. And, in all other cases, the store or load would proceed without being redirected. This is illustrated in Figure 6.1.

Notice that the design could provide the desired safety guarantee (that, while executing shadow code, speculating threads modify only shadow memory) without adding any calls to the load check support routine. However, in such a design, speculating threads may often issue load instructions that, by loading from non-shadow memory regions rather than copies of those non-shadow shadow regions, obtain data values that are "stale" with respect to speculative execution. Therefore, not adding calls to the load check support routine may unnecessarily introduce incorrect data values into speculative execution, which may hurt the design's ability to deliver performance benefit.

The SpecHint design makes one further specification to avoid unnecessarily introducing incorrect data values into speculative execution as a side effect of implementing copy-on-write between threads in the same address space rather than different processes. In particu-

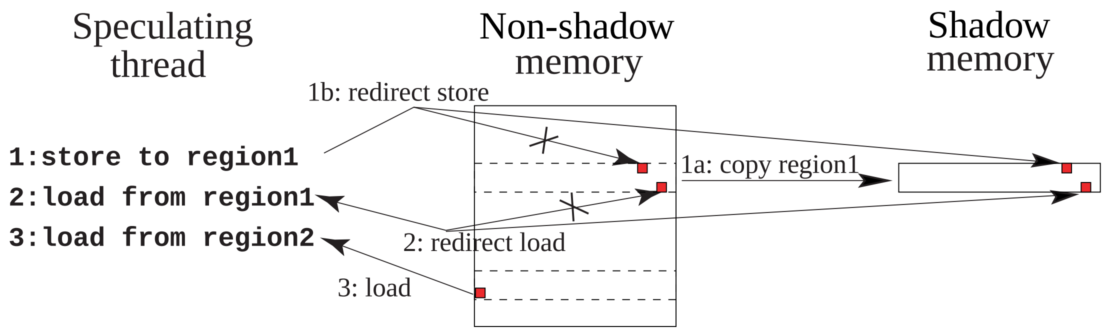

Figure 6.1: Illustration of how software copy-on-write works. (1) The first time the speculating thread attempts to store to an uncopied non-shadow memory region, the region is copied and the store is redirected to the copy. (2) Loads from (and stores) to copied regions are redirected to the copy, while (3) loads from uncopied regions are allowed to proceed without redirection.

lar, notice that, after a copy-on-write page shared by two processes is copied, each process can map its own copy of the page at the address where it had original mapped the shared page. However, after a copy-on-write region shared by two threads in the same process is copied, the copy of the region must reside at a different address than the original region. This could unnecessarily introduce incorrect data values into speculative execution as illustrated in Figure 6.2. In this example, buf is an array which happens to cross a region boundary; its first element is in region A at address n, and its second element is in region B at address $n + 1$. Consider what could happen when a speculating thread executes code that, if executed by an original thread, would modify the values of each of these elements in turn. The speculating thread would pass the address n to the store check support routine, which would return p, the address of the copy of the first element. If the speculating thread modifies its value for buf to be p (such that any future attempts to access the first element of buf would not need to be redirected) then it will believe, incorrectly, that the second element is at address $p+1$. This could, for example, cause it to overwrite unrelated shadow memory (i.e. if $p + 1$ is in an unrelated shadow memory region). (Notice that it would not allow the speculating thread to modify non-shadow memory.) Fortunately, the design can avoid unnecessarily introducing incorrect data values into speculative execution in this fashion by guaranteeing that a pointer to some data in a non-shadow memory region will never be changed to point instead to the copy of that data (in a shadow memory region). Since all addresses in copies of non-shadow memory regions are initially obtained by calling the store or load check support routine, the SpecHint design provides this guarantee by specifying that the way in which shadow code is modified ensure that speculating threads can use an address returned by a call to the store or load check support routine to update a value in shadow memory only indirectly (i.e. by accessing the data at that address).

The idea of enforcing memory protection in software instead of hardware by adding checks before load and store instructions is not new. Simpler checks have been used to implement software fault isolation [67]. Notice that one potential advantage of implementing

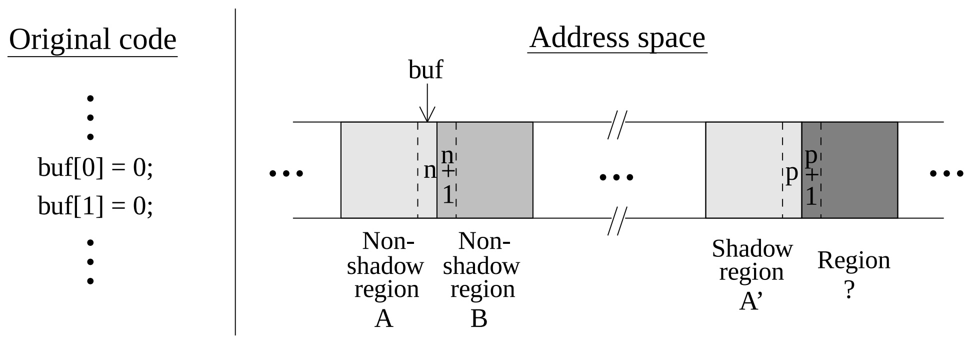

Figure 6.2: An example of how updating a pointer to an object in non-shadow memory to point instead to the copy of the object in shadow memory could cause incorrect data values to be introduced unnecessarily into speculative execution. In particular, when speculative execution executes the shadow code version of this original code, it will execute a store check before updating each element of `buf`. Upon executing the first store check, it will learn that the copy of `buf` is at address `p`. If it updates its value for `buf` to be `p` rather than `n`, however, then it will subsequently miscalculate the address of the second element in `buf`.

copy-on-write in software rather than using standard copy-on-write is that, while, on most systems, the granularity of standard copy-on-write is fixed to be the system's page size, a software implementation of copy-on-write could choose any region size and could even choose to vary region sizes. On the other hand, software implementations will tend to incur more overhead. For example, the main disadvantage of including software copy-on-write in the SpecHint design is that performing copy-on-write checks will add a large amount of work to speculative execution, which may hurt the design's ability to deliver performance benefit as discussed in Section 6.2.

Notice that basing a speculative execution design on selective copy-on-write rather than ordinary copy-on-write could affect the performance of the design in two ways. First, it may reduce the overhead of the design. In particular, if a design uses ordinary copy-on-write, the first execution that attempts to modify a copy-on-write unit would suffer the cost of copying the unit. On the other hand, a design that uses selective copy-on-write could guarantee that normal execution could modify copy-on-write units without suffering the cost of copying those units by not including normal execution in the subset of entities that can, by modifying copy-on-write units, cause those units to be copied. This may not be a noticeable benefit, however, since the cost of copying a copy-on-write unit may be negligible. For example, on my evaluation machine, modifying a copy-on-write page takes only around 165 microseconds.

Second, consider how selective copy-on-write affects speculative execution's *effective address space* (the subset of its address space that it can access which, depending on the design, may be its entire address space but, for the SpecHint design, is shadow memory plus any uncopied non-shadow memory regions while the speculating thread is executing shadow code). Notice that, until speculative execution modifies a given copy-on-write unit, any changes that normal execution makes to that unit will automatically be included in speculative execution's effective address space. Therefore, basing a design on selective rather than ordinary copy-on-write will sometimes enable speculative execution to leverage runtime values determined during normal execution without resynchronizing with normal execution, which may be an advantage since resynchronizing with normal execution will incur some cost. On the other hand, basing a design on selective copy-on-write may cause speculative execution to see inconsistent memory values (where, by "inconsistent", I mean that speculative execution's effective address space may include a value that normal execution set at some particular time without including all values that normal execution set before that time). In particular, if speculative execution modifies some copy-on-write unit, and then normal execution modifies first that unit and then some other unit that has not yet been copied, then speculative execution's effective address space will include the latter update but not the former update. It is unclear whether the combination of these two effects would improve, degrade, or have no noticeable effect on the performance of a design.

The SpecHint implementation

The SpecHint implementation satisfies the specifications in this section. The SpecHint tool uses simple mechanisms to reduce the amount of work added to speculative execution for the purpose of supporting software copy-on-write.

As background, on the architecture targetted by my implementation (the Alpha architecture [50]), a store instruction designates the value to store and the address at which to store it, where the address is the sum of a register value and a constant offset. A load instruction designates the address from which to load a value, where the address is specified as for a store instruction, and the register in which to place that value. I will refer to the register whose value is used to determine the address specified by some load or store instruction as the "base register" and will say that the instruction is a load or store "off" this base register (e.g. "loads off the stack pointer" are load instructions whose base register is the stack pointer). The architecture limits the range of the constant offset that can be specified to ± 32 KB.

The SpecHint tool modifies shadow code such that the speculating thread will perform a copy-on-write check before every load and store instruction in shadow code, except loads and stores off the stack pointer or global pointer. (The global pointer is an Alpha-specific register commonly used as the base register for accessing global data.) The tool guarantees that copy-on-write checks need not be performed before loads and stores off the stack pointer by ensuring that those loads and stores allow speculating threads to access only memory allocated for their stack. In particular, the tool guarantees that, while speculating threads are executing shadow code, their stack pointer always contains an address that is in the memory allocated for their stack, but not in the top or bottom 32 KB of that memory. This is sufficient because, as discussed in the previous paragraph, loads and stores can access only memory within 32 KB of the address in their base register. This is implemented in the simplest possible manner, i.e. by adding a check before every instruction that could change the stack pointer (as described in Section 6.3.1.3). Eliminating the need for copy-

Benchmark	% of instructions which were			
	Load/stores off		Other	Stack pointer
	Stack pointer	Global pointer	loads/stores	changes
Agrep	2	0	29	0
Gnuld	8	1	17	2
XDataSlice	17	0	20	2
Postgres 80%	13	3	12	4
Postgres 20%	13	2	11	4
Sphinx	5	1	16	1

Table 6.1: To estimate the benefit of not adding copy-on-write checks for loads and stores off the stack pointer (and global pointer), and the amount of work that the remaining copy-on-write checks will add to speculative execution, I used pixie [51] to measure dynamic instruction counts while executing my benchmarks using original application binaries. (The benchmarks are described in Section 7.3.) The second and third column estimate how much more frequently copy-on-write checks would be performed if they needed to be performed before loads and stores off the stack pointer and global pointer, respectively. The fourth column estimate how frequently copy-on-write checks would still need to be performed. The last column shows how frequently stack pointer checks would be performed.

.

on-write checks before loads and stores off the global pointer is a small and undescribed optimization. Table 6.1 shows, for executions of unmodified application binaries, percentages of instructions executed which are loads or stores off the stack pointer, loads or stores off the global pointer, other loads or stores, and instructions that change the stack pointer. These figures indicate that eliminating copy-on-write checks before loads and stores off the stack pointer can substantially decrease the number of copy-on-write checks performed, even when accounting for the addition of stack pointer checks (which are simpler and required for another reason anyhow, as discussed in Section 6.3.1.3). However, the remaining copy-on-write checks will probably still add a lot of work to speculative execution.

6.3.1.2 System calls

The SpecHint design restricts the system calls that speculating threads can issue while executing shadow code in the following two ways. First, it specifies that the binary modification tool remove from shadow code all *trap instructions* (instructions that could initiate a system call) that could be used to issue a system call that could fail to return to user-level code (e.g. the $exit$ system call). Second, it specifies that the tool remove from shadow code all trap instructions that could change any state (internal or external to the process) other than the calling thread's register values, except if: 1) the trap instruction could be used only to issue a system call that would, in addition, change at most the contents of one or more input buffers specified in arguments to the call, and 2) the binary modification tool ensures that all buffers so specified are entirely contained within shadow memory. Section 6.3.2.2

will discuss how the design ensures that shadow memory does not overlap pages mapped for original execution, such that this specification is sufficient to guarantee that speculating threads, while executing shadow code, cannot produce direct or indirect output by issuing system calls. It also guarantees that, while executing shadow code, speculating threads cannot (for example) map devices, change their signal mask, change the actions specified for handling signals, or issue system calls that might map pages.

Notice that the combination of software copy-on-write, and preventing system calls that might map pages, guarantees that speculating threads cannot in any way claim an additional piece of the address space while executing shadow code. In particular, a speculating thread could call the shadow code version of some user-level memory allocation routine (e.g. $malloc$), but software copy-on-write will prevent it from updating whatever data structure the original code version of that routine uses to designate some memory as allocated (as well as preventing it from modifying the memory at whatever address the routine returns, unless that address is in shadow memory). To allow the speculating thread to allocate memory that it may need to generate accurate prefetches, the design specifies that the binary modification tool should replace any calls in shadow code to recognized memory allocation routines (e.g. the memory allocation routines in the standard C library) with appropriate calls to the shadow allocation support routines (discussed in Section 6.3.2.2).

The SpecHint implementation

The SpecHint tool satisfies this specification by removing all trap instructions from shadow code, except for the ones in the wrapper routines for the $stat$ and $lstat$ system calls (both of which are commonly used to obtain information about a file). Issuing one of these system calls can change only register values of the calling thread and the contents of a buffer specified in the arguments to the call. The SpecHint tool guarantees that the specified buffer is entirely contained within shadow memory. Figure 6.3 illustrates how, to provide this guarantee, the SpecHint tool modifies the shadow code version of the wrapper routine for the $stat$ system call.

6.3.1.3 Signal masks and stack space

This section describes how the SpecHint design guarantees that, at all times after being initiated, speculating threads will block all but exception signals. For the reasons discussed in Section 6.2.2, this should be sufficient to ensure that they will not produce direct or indirect output by receiving asynchronous signals. It also describes how, based on an additional assumption, the SpecHint design guarantees that speculating threads, while executing shadow code, will always have enough space in the stack(s) that would be used for signal handling to handle any signals they might receive. This is important because, in order to prevent speculating threads from producing direct or indirect output by triggering exceptions, the action specified for exception signals will always be to execute a signal handler. If, however, the stack that would be used to execute this handler has insufficient space when the

```
stat(char *path, struct stat *buffer)    shadow_stat(char *path, struct stat *buffer)
                                           sub  sp, 8+SIZEOF(struct stat), sp
                                           st   a1, 0(sp)
                                           add  sp, #8, a1
   add  zero, #67, r0                      add  zero, #67, r0
   trap                                    trap
   beq  r19, success                       beq  r19, success
                                           add  sp, 8+SIZEOF(struct stat), sp
   ... error handling code ...             ... error handling code ...
 success:                                success:
                                           ld   a0, 0(sp)
                                           add  sp, #8, a1
                                           add  zero, #SIZEOF(struct stat), a2
                                           call returnreg, shadow_memcpy
                                           add sp, 8+sizeof(struct stat), sp
   ret (returnreg)                         ret (returnreg)
```

Figure 6.3: How to ensure the safety of the `stat` system call in shadow code, which may update a memory area indicated by the `buffer` argument. Simply replacing `buffer` with the corresponding pointer into shadow memory would not always be sufficient because the `sizeof(struct stat)`-sized buffer area may cross a region boundary. If so, the data returned by the `stat` call may overwrite non-shadow memory since no assumptions can be made about the relative positions of copy-on-write region copies. A simple, safe alternative is to use the speculating thread's stack for temporary contiguous memory in shadow memory. Notice that `shadow_memcpy` will trigger software copy-on-write as appropriate if any part of the memory area indicated by the original `buffer` argument is in non-shadow memory. Finally, the initial stack pointer adjustment is preceded by a check (as discussed in Section 6.3.1.1) that is omitted for simplicity.

operating system attempts to deliver an exception signal to the speculating thread, then the operating system will terminate the process (as discussed in Section 6.2.2).

This SpecHint design guarantees that, at all times after being initiated, speculating threads will block all signals except exception signals in the following manner. First, it specifies that the binary modification tool ensure that any system calls that the original thread would issue to change the process-wide signal mask become instead system calls to change only the original thread's signal mask. On most operating systems, this cannot change the output of normal execution. Next, it specifies that, when a speculating thread is spawned, its designated startup routine will be the *thread init support routine* (one of the support routines for speculating threads) and that this routine will begin by setting the speculating thread's signal mask to block all but exception signals (and end by calling the resynchronization routine). Finally, it guarantees that there is no other trap instruction in all the code which the speculating thread can execute that could allow it to issue a system call which could change its signal mask. It provide this guarantee by specifying that the support routines for speculating threads and the exception handling support routine contain no other trap instructions that could be used to change the speculating thread's signal mask, and guaranteeing (as discussed in Section 6.3.2.3) that the speculating thread can execute only support routines for speculating threads, the exception handling support routine, and shadow code. This is sufficient because, as discussed in the previous section, the design also guarantees that shadow code can contain no trap instructions which could be used to change the speculating thread's signal mask.

The stack(s) that would be used to handle signals will be either the speculating thread's stack, or a specified alternate stack. The SpecHint design does not allow speculating threads or added normal execution to change the designation of an alternate stack, so an alternate stack will only be specified due to the original normal execution. I will refer to the amount that the operating system requires to deliver a signal to an empty signal handler (a signal handler that simply returns) as $OSHandleSpace$. The SpecHint design is based on the following additional assumption:

Assumption 6. If original normal execution specifies an alternate stack for signal handling, then there will be enough (i.e. $OSHandleSpace$) read-able and write-able space at the bottom of that stack for the operating system to deliver a signal.

I claim this is a reasonable assumption for a broad class of applications because, otherwise, an exception would be triggered whenever the operating system attempted to deliver a signal on this stack, to the original normal execution as well as to speculative execution. This would defeat the purpose of the original normal execution specifying an alternate stack. Based on this assumption, the SpecHint design guarantees that there will always be enough space in any specified alternate stack to handle any signals the speculating thread might receive by guaranteeing that the maximum amount of space needed to handle any such signals will be $OSHandleSpace$. (How the design provides this guarantee is discussed in Section 6.3.2.1.)

The SpecHint design guarantees that, while executing shadow code, speculating threads will always have enough space in their own stack to handle any signals they might receive by guaranteeing (as discussed in Section 6.3.2.3) that the speculating threads can only enter shadow code from one of the support routines for speculating threads, specifying that the support routines for the speculating threads ensure that there is at least $OSHandleSpace$ stack space available whenever they allow speculating threads to enter shadow code, and guaranteeing that the speculating thread will never, while executing shadow code, be able to change its stack pointer such that it will have less than $OSHandleSpace$ available in its stack. The design provides the last guarantee by specifying the behavior of a type of check I will refer to as a *stack pointer check*, and what the binary modification tool must ensure in modifying shadow code to include sufficient stack pointer checks. In particular, the design specifies that the *safe stack area* be the memory allocated for the speculating thread's stack, excluding at least $OSHandleSpace$ at the top of the stack. (At the end of this section, I give an example of why an implementation might decide to exclude more of the stack.) The design specifies that the binary modification tool modify shadow code such that a thread which executes shadow code will perform a stack pointer check before every instruction that might otherwise change the value of the stack pointer to an address not in the safe stack area. Finally, the design specifies that a stack pointer check abort the current speculation if the address this instruction would change the stack pointer to is not in the safe stack area.

The SpecHint implementation

The SpecHint implementation satisfies the specifications in this section. The SpecHint tool makes no attempt to reduce the number of stack pointer checks; that is, it adds a stack pointer check before every instruction that could change the stack pointer. Table 6.1 shows how frequently such instructions occur during executions of some unmodified application binaries. With my implementation and target system (Digital UNIX 3.2), $OSHandleSpace$ is only 4 KB. For the reasons discussed in Section 6.3.1.1, in order to reduce the number of copy-on-write checks the tool adds to shadow code, the SpecHint tool defines the safe stack area to also exclude the top and bottom 32 KB of the memory allocated for the speculating thread's stack, and my implementation of the support routines for speculating threads ensures that the speculating thread's stack pointer will always contain an address in this safe stack area upon entering shadow code.

6.3.1.4 Control transfers

This section describes how the SpecHint design guarantees that speculating threads will be able to escape shadow code only by either calling a support routine for the speculating thread or receiving an exception signal. This is important because it allows the design to limit what code the speculating thread could possibly execute (and, therefore, the ways in which it could produce direct or indirect output).

If a thread is executing in some code that fills one or more contiguous address ranges, then it can escape that code only by either executing a *control transfer* (an instruction that could cause a non-sequential change in the program counter), or executing sequentially past the last instruction in one of those ranges. A control transfer is direct/indirect if its target address is determined by a constant/register value. Thus, direct control transfers always have only one target address, and it is always possible for a binary modification tool to determine (and adjust) that target address. On the other hand, it may not be possible for a binary modification tool to determine the possible target address(es) of indirect control transfers.

The SpecHint design guarantees that speculating threads will be able to escape shadow code only by either calling a support routine for the speculating thread or receiving an exception signal, in the following manner. First, it specifies that the binary modification tool insert a call to the resynchronization support routine (which never returns) after the last instruction of all address ranges that contain shadow code for which the last instruction may allow a thread to advance to the next address. Second, it specifies that the binary modification tool ensure that all direct control transfers have, as their target address, either an address in shadow code or the entry address of one of the support routines for speculating threads. Finally, it specifies the behavior of *control transfer checks* and the check routine called by control transfer checks, the *control transfer support routine*, and what the binary modification tool must ensure in modifying shadow code to include control transfer checks. In particular, to satisfy this guarantee, the SpecHint design specifies that the set of *valid target addresses* exclude at least all addresses not either in shadow code or the entry address of one of the support routines for speculating threads that the design specifies to be callable from shadow code.[1] (In the latter half of this section, I discuss why an implementation would probably exclude many addresses in shadow code as well.) The design specifies that the control transfer support routine behave as follows:

- It should expect an address as its argument.
- If the argument specifies a valid target address, then it should either return that address, or abort the current speculation.
- If the argument specifies an address that is in original code (as opposed to shadow code), and the corresponding address in shadow code is a valid target address, then it should either return that address in shadow code or abort the current speculation.
- Otherwise, it should abort the current speculation.

(The design further specifies that the behavior of this support routine be restricted as described in Section 6.3.2.3.) In addition, the design specifies that the binary modification tool modify shadow code to include sufficient control transfer checks. In particular, the design specifies that the binary modification tool modify shadow code such that a thread which executes shadow code will call (with the correct return address) the control transfer support routine before every indirect control transfer that might otherwise cause the program counter to change to an address which is not a valid target address, specifying what would be the target address of the control transfer as the argument to the routine, and will then use the address specified as the routine's return value as the target address of its control transfer instead of the address it specified as the argument to the call. Notice that a binary modification tool could meet this specification by modifying shadow code such that speculating threads will execute a control transfer check before every indirect control transfer in shadow code. Notice also that, since shadow code is a copy of the original code, the target addresses of most indirect control transfers will be in original code. Furthermore, as discussed in Section 4.2.3, because the speculating thread's stack is copied from the original thread's stack during resynchronization, some return addresses in the stack may specify addresses in original code. However, prior work on binary modification [29] indicates that it would be possible for a binary modification tool to determine and appropriately adjust the target addresses of almost all indirect control transfers other than returns that may use addresses copied from the original thread's stack. This would reduce the amount of work added to speculative execution.

[1]The design specifies that only the following support routines are callable from shadow code: the resynchronization support routine (discussed in Section 6.3.1), the check support routines, the shadow allocation support routines (discussed in Section 6.3.2.2), and the I/O support routines (discussed in the next chapter). The design specifies that the thread init support routine (discussed in the previous section) can be called only as the startup routine for the speculating thread, and the remaining support routines for speculating threads can be called only from other support routines for speculating threads.

The SpecHint implementation

The SpecHint implementation satisfies the specifications in this section. To reduce the number of necessary control transfer checks, the SpecHint tool adjusts all indirect control transfers whose target addresses it can determine such that their target addresses are the appropriate addresses in shadow code rather than original code. However, while (for my benchmarks) it is able to determine the target addresses of almost all indirect calls, it is not able to determine all possible target addresses of most returns (indirect jumps). In particular, it allows the speculating thread to store and load return addresses in/from the stack, but the tool does not perform the necessary data flow analysis to ensure that the speculating thread cannot corrupt those stack values. Previous work [29] has demonstrated that this would be possible.

The SpecHint tool greatly restricts the set of valid target addresses because it also uses control target checks to ensure that the checks it inserts into shadow code will be executed when they ought to be executed. To elucidate, control is said to "flow" from one instruction to another when the second instruction is executed immediately after the first instruction. In order to meet the SpecHint design's specification about when the various checks must be executed, any binary modification tool needs to ensure that control cannot flow in a way that would allow a speculating thread to either skip a check, or jump to an instruction that would cause the check to behave other than as intended – e.g. by jumping into an address in the middle of a check or, since checks often rely on one or more addresses being loaded correctly (like the address of a check support routine), jumping to any address that would allow the thread to load incorrect addresses during a check. The SpecHint tool modifies shadow code such that, as long as all the indirect control transfer whose target addresses it could not determine could flow only to a particular subset of addresses, then such misbehavior could not occur. This subset includes all the (adjusted) target addresses of indirect control transfers whose target addresses it could determine, plus all addresses right after calls that are not in checks (i.e. all expected return addresses, except the ones from calls to check support routines since those addresses are always in the middle of a check). Therefore, to satisfy the specifications of the SpecHint design, the SpecHint tool needs to add control transfer checks only before indirect control transfers whose target addresses it is unable to determine. In addition, to allow the control transfer support routine to detect some (though not all) addresses in this restricted set of valid target addresses, the SpecHint tool adds an initialized global data structures to the transformed binary which allows the control transfer srpport routine to detect, and map from original code, entry addresses of functions in shadow code, and valid return addresses in shadow code. (Mapping from return addresses in original code to return addresses in shadow code is necessary to repair return addresses stored in the stack, as discussed in Section 4.2.3.)

6.3.1.5 Discussion

This section discusses the advantages and disadvantages of shadow code. Note that I do not claim that shadow code is the optimal design choice. The purpose of this section is simply to point out what the trade-offs might be between adding (and modifying) a complete copy of the code, and modifying the original code to include the checks discussed during this section such that the speculating thread could execute the original code. Notice that, since the original thread should not perform any of these checks, modifying the original code would also require adding a conditional branch before every check, that would cause the original thread to skip the checks.

Adding a complete copy of the code has three potential disadvantages: 1) it will increase the size of the binary by a greater amount, such that the binary requires more storage space, 2) it may cause an execution of the binary to claim more physical memory in order to hold both original and shadow code pages, and 3) it may cause speculating threads to experience more page faults since the code pages they will need to execute will differ from the code pages normal execution will bring into memory while it is executing. The first may be a minor disadvantage in most cases since disk space is inexpensive and often underutilized. The second is more problematic because, as discussed in the previous chapter, increasing contention for memory can hurt the performance of normal executions. I observe anecdotally, however, that code pages are usually a small fraction of the pages in physical memory; physical memory tends to be filled mostly with heap pages and file pages. Therefore, minimizing the number of code pages may not be of primary importance. Finally, the third point will at most hurt the performance benefit of the design. My evaluation results show that the performance benefits of my implementation of this design are substantial, such that this design, while perhaps not optimal, is sufficient to answer the main questions I set out to answer in this dissertation work (specifically, whether the speculative execution approach, and an implementation that requires no operating system modifications specific to this approach, have the potential to deliver substantial performance benefits).

In contrast, modifying the original code to include the same set of checks, plus an additional conditional branch before each check that causes the original thread to skip the check, has the following two potential disadvantages: 1) it will add work to normal execution because normal execution will need to branch around each check, and 2) it may cause normal execution to experience more page faults because the code it needs in order to execute will now be spread out over more pages. Both of these are problematic because they will add overhead to normal execution. In particular, from the figures in the last column of Table 6.1, these extra conditional branches might increase the number of instructions executed during normal execution by over 20%. While this would not matter while executing an application that spends most of its execution time stalled on I/O, it violates the design goal of low overhead. For example, it could substantially hurt the performance of an application whose data is sometimes in memory. It would also make the design more risky for application that spend a somewhat smaller fraction of their execution time stalled on I/O, but might still benefit noticeably from I/O prefetching. Furthermore, there is a disadvantage, from an implementation standpoint, to inserting instructions into original code in a manner that changes the addresses of most instructions. In particular, the binary modification tool would need to guarantee that all control transfers were properly adjusted (either during modification or while executing the modified binary) to reflect those address changes. This differs from the adjustment of control transfers in shadow code because shadow code is only executed by the speculating thread, and it is always reasonable to simply abort the current speculation.

6.3.2 Support routines

The support routines provide functionality essential to the design that is not specific to any particular application. The support routines can be divided, based on their function, into the routines focused on enforcing safety, the routines focused on generating prefetches, and the resynchronization support routine (which is essential to both functions). The previous section specified a subset of the routines focused on enforcing safety, the check support routines, and some behavior of the resynchronization support routine. This section specifies the other routines focused on enforcing safety, additional behavior of the resynchronization support routine, and the general restrictions that the design imposes on sets of support routines in order to allow it to provide, with two additional assumptions, the sets of guarantees discussed in Section 6.2 as being sufficient for acceptable safety. (Since the next chapter focuses on the issues in developing a design that will generate accurate and timely prefetches, the behavior of the resynchronization support routines that is focused on generating prefetches, and (aside from the general restrictions specified in this section) the support routines focused on generating prefetches, are more properly specified in the next chapter.)

Section 6.3.2.1 describes specifications for ensuring that the handling of signals cannot produce direct or indirect output. Section 6.3.2.2 then describes specifications for ensuring, that the speculating thread and added normal execution allocate and modify only memory that does not overlap any pages mapped for original execution. Finally, Section 6.3.2.3 describes how the design restricts support routines.

6.3.2.1 Signal handling

This section describes how the design guarantees that the maximum amount of space needed to handle any signals delivered to the speculating thread will be the maximum amount required by the operating system to deliver a signal (which, as discussed in Section 6.3.1.3, assists the design in guaranteeing that there will always be enough stack space to handle any signals speculating threads may receive). It also describes how the SpecHint design guarantees that the speculating thread cannot produce direct or indirect output by receiving exception signals.

First, the SpecHint design specifies that the binary modification tool add an uninitialized global data structure I will refer to as the *handling array*. Each entry in this array will be used to contain values indicating what action would have been specified for a particular exception signal type, and whether it would have been specified that the default action should be restored as soon as one such signal is received. The entry should have one additional entry that can be used to hold the same information about a prospective change during a system call that may change the action specified for a signal. Second, the SpecHint design specifies that the exception handling support routine behave as follows when executed as a signal handler:

- It requires no stack space.
- When executed by the speculating thread, it simply ensures that, when the speculating thread next executes user-level code, it will start executing from the entry address of the resynchronization support routine. It should accomplish this in a manner that cannot trigger any exceptions, produces no direct or indirect output, and calls no other routines.
- When executed by the original thread, it uses the values in the appropriate entry in the handling array to cause the same effect as would have occurred had the exception handling support routine not been designated as the specified signal handler (as discussed in Section 6.2.2). If the entry indicates that the default action would have been restored, it updates that entry to indicate that the specified action would, in future, be to terminate the process, and to indicate that the default action no longer needs to be restored.

Third, the SpecHint design guarantees that the action specified for exception signals received by speculating threads will always be to execute the exception handling support routine without resetting the signal action. The SpecHint design provides this guarantee by specifying three support routines for the original thread and when they should be called by the original thread. The *thread spawn support routine* should be called shortly after normal execution begins. It should discover the current action specified for each exception signal, use this information to update the handling array, and then change the action specified for each exception signal to executing the exception handling support routine, without resetting the handler. Next, the *pre-signal action support routine* should be called before the original thread issues a system call that may change the specified action for an exception signal, except when this system call is being performed on behalf of the thread spawn support routine. It should update the handling array entry for a prospective change, and then change the parameters to the call such that they will instead specify that the action should be to execute the exception handling support routine without resetting the signal action. Finally, the *post-signal action support routine* should be called after the original thread returns from a successful system call to obtain or change the specified action for an exception signal, except when this system call is being performed on behalf of the thread spawn support routine. If the call obtained the previous action, it should modify that information based on the handling array entry for the appropriate signal type so that the returned action will be what it would have been without support for speculative execution. In addition, if the call changed the specified action, then it should reflect that in the handling aray by copying the handling array entry for a prospective change to the entry for this signal type.

Since speculating threads block all signals but exception signals (as discussed in Section 6.3.1.3), the third point, plus the specifications that the exception handling support routine require no stack space and trigger no exceptions, are sufficient to guarantee that the maximum amount of space needed to handle any signals delivered to the speculating thread will be the maximum amount required by the operating system to deliver a signal. Moreover, assuming (as provided for in Section 6.3.1.3) that there will always be sufficient stack space to handle any exception signals the speculating thread receives, these are sufficient to ensure that the speculating thread cannot produce direct or indirect output by receiving exception signals. Finally, for the reasons discussed in Section 6.2.2, these should be sufficient to ensure that, in changing the actions specified for exceptions signals, added normal execution would not produce unsafe changes in the output of a broad range of applications.

The SpecHint implementation

The SpecHint implementation is not a full implementation of this specification in two ways. It does not keep track of whether actions should be reset, and it does not update returned information about the previous signal action. None of my benchmark applications specified that any signal handlers should be reset to their default, or used information about the previous signal action.

6.3.2.2 Memory allocation

The SpecHint design relies on two additional assumptions. First, recall that, as discussed in Section 6.2.2, any execution can map any page by demanding that page with the appropriate $mmap$ or $shmat$ system call, even if the page was already mapped. I will refer to mapping requests that could cause the operating system to map pages that are already mapped, unmapping whatever was in those pages, as *preemptive mapping requests*. Preemptive mapping requests pose a problem for designs based on binary modification because they could potentially cause the unmapping of any subset of the code and/or global data on which such a design must rely.[2] Thus, the SpecHint design relies on the following additional assumption:

Assumption 7. Original normal execution will not preemptively map a page that holds code or global data added by the binary modification tool, or any part of the shadow thread's stack.

Second, the design maps pages only in two ways: via the system call to spawn the speculating thread (which will cause the operating system to map pages to hold a stack for the speculating thread), and via $mmap$ system calls that do not demand particular pages. However, the design does not ensure that these pages will be unmapped before their being mapped interferes with the growth of the primary thread's stack or the brk heap. In particular, the design relies on the following additional assumption:

[2]The safety of the SpecHint design also depends on no modifications being made to the code added by the binary modification tool. Notice that original normal execution will not modify this code based on Assumption 1 from Section 6.2.2. Software copy-on-write ensures that the speculating thread will not modify this code while executing shadow code. Finally, the restrictions on all other code (e.g. support routines) added by the binary modification tool (as detailed in the next section) ensure that this code will not be modified by either added normal execution or the speculating thread while it is not executing shadow code. This does, however, mean that this design may be ineffective (but safe) for applications whose data needs depend on dynamically generated code.

Assumption 8. The pages that operating systems will map in response to mapping requests that do not specify particular pages will rarely interfere with the growth of the primary thread's stack or the brk heap.

I claim that this is a reasonable assumption because, in order to decrease the likelihood that mapped pages interfere with the growth of the primary thread's stack or the brk heap, operating systems will automatically map pages far from the primary thread's stack or the brk heap when mapping requests do not demand particular pages. Since address spaces are so large, it is rare for pages so mapped to interfere with the growth of those memory regions.

The design specifies that the binary modification tool add code and global data to the transformed binary such that they will not overlap any pages already claimed in the original binary, and the pages they use are unlikely to interfere with the growth of the primary thread's stack or the brk heap. The design guarantees that the speculating thread's stack does not overlap any pages mapped for original execution by specifying some behavior of the thread spawn support routine. In particular, it specifies that this routine should (after initializing signal handling as described in the previous section) spawn the speculating thread with attributes indicating that its stack should be surrounded by a guard page on either side. (A guard page is an inaccessible page located on either side of some data structure.) This will cause the operating system to map a set of contiguous pages for the speculating thread's stack, such that the stack will not overlap any previously mapped pages.

Assuming that speculating threads can only allocate memory by calling allocation routines in the memory allocation support package (which is provided for in Section 6.3.2.3), the design guarantees that memory allocated by speculating threads cannot overlap any pages mapped for original execution by specifying the behavior of the *memory allocation support package*, which contains the shadow allocation support routines, and some support routines for original threads. In particular, the design specifies that the memory allocation support package behave as follows:

- The package maps pages by issuing $mmap$ system calls that do not demand particular pages.
- The package includes a routine that will explicitly unmap all pages it mapped since the last time it executed this routine (but not any other pages), by issuing $munmap$ system calls. I will refer to this set of pages as the pages "currently managed by" the package. This routine also updates all roots (in global data) of data structures allocated by support routines for speculating threads to reflect the fact that those data structures are now empty (such that the support routines will rebuild those data structures from scratch). This routine can be called only from the resynchronization support routine, such that it can be considered part of the resynchronization support routine.
- The package divides memory on pages it is currently managing into two pools: the *shadow memory pool* and the *non-shadow memory pool.*
- When a shadow/other allocation routine is called to obtain some amount of memory, it either returns a pointer to a contiguous unused memory buffer of the correct size

that is entirely contained on pages in the shadow/non-shadow memory pool (updating the package's notion of what memory is unused and what buffers are in use), or aborts the current speculation. And,

- When a shadow/other allocation routine is called to release memory, if the address provided specifies an in-use memory buffer in the shadow/non-shadow memory pool, then it should update the package's notion of what memory is used and unused. Otherwise, it should simply return.

The non-shadow memory pool is intended for data structures allocated for use internal to the support routines for speculating threads (e.g. whatever data structure the copy-on-write support routines use to maintain the mapping between non-shadow memory regions and their copies in shadow memory). Notice that, since shadow memory consists of the speculating thread's stack and memory allocated via a shadow memory allocation routine, the copy-on-write checks will ensure that the speculating thread cannot modify pages in the non-shadow memory pool while executing shadow code.

The design also specifies some behavior of the resynchronization support routine as follows. Upon entering the resynchronization support routine at its entry address, the speculating thread will immediately call the routine which explicitly unmaps all pages currently managed by the memory allocation support package. Subsequently, it will not proceed beyond a certain point in the routine, i.e. before all actions that might result in a memory allocation call, until the original thread is not in the process of performing a check that I refer to as a $preemptive mapping check$. The design specifies the behavior of preemptive mapping checks, and when they should be performed, as follows. The design specifies that the binary modification tool ensure that original threads will perform a preemptive mapping check whenever they execute a trap instruction that may cause the operating system to unmap a mapped page. The design further specifies that the tool will ensure that speculating threads cannot have an $mmap$ call in progress while such a check is in progress. The design specifies that the check cause the original thread to behave as follows. If the trap instruction may cause the operating system to unmap a page currently managed by the memory allocation support package, then, before executing the trap instruction, the original thread will send an exception signal to the speculating thread and then block until the speculating thread has unmapped all pages currently managed by the memory allocation support package. The check is said to have ended after the original thread returns from the trap instruction.

Since (as discussed in Section 6.3.1.3 and the previous section) the design guarantees that the speculating thread will not block exception signals, and will enter the resynchronization support routine at its entry address after receiving an exception signal, the specifications above for the resynchronization support routine guarantee that generating an exception signal for the speculating thread will result in the speculating thread unmapping all pages currently managed by the memory allocation support package. Therefore, the specifications above regarding preemptive mapping checks ensure that the memory allocation support package will unmap all pages it is managing if the original thread is about to preemptively map one or more of those pages, such that the package will never mistakenly believe that it is managing a page that is actually mapped for original execution. Finally, by forcing the speculating thread to switch from whatever code it was executing to the resynchronization support routine, these specifications ensure that the speculating thread could not modify a page mapped for original execution due to a race condition between the speculating thread and the original thread.

Notice that the design causes the speculating thread to release all mapped pages every time it resynchronizes with the original thread. In addition to relying on this behavior in ensuring that preemptive mapping requests for pages allocated by the speculating thread do not enable the speculating thread to modify pages mapped for original execution, the design also uses this mechanism to prevent the speculating thread from introducing memory leaks. Moreover, it depends on this mechanism to reduce the likelihood that the speculating thread will hog so much memory that the operating system would refuse mapping requests from the original normal execution. The design could easily be modified to further reduce the likelihood that the design could cause original normal execution to fail mapping requests by ensuring that, if a mapping request from original normal execution is refused, the original thread will send a signal to the speculating thread, block until the speculating thread has released its pages, and then issue a new mapping request (i.e. something similar to what is described above). Entirely eliminating the possibility that the design could cause normal execution to fail a mapping request may not be feasible, however, because this could require unmapping all the code and global data added by the binary modification tool as well, and the design relies on the continued presence of that code and global data.

The SpecHint implementation

The SpecHint implementation does not meet this specification in a number of ways. First, it does not check for preemptive mapping requests by original normal execution. None of my benchmark applications issued any preemptive mapping requests. Second, it does not ensure that the code and global data it adds will not overlap any pages claimed in the original binary. Finally, the memory allocation support package issues calls to the standard library dynamic allocation routines (e.g. $malloc$) rather than issuing $mmap$ calls. This did not present a problem for my benchmark applications because they all used the standard library dynamic allocation routines, and their original normal executions only accessed memory (versus pages) that contained original code or original global data, and memory allocated via these allocation routines. I claim that the latter is the case for all well-written applications, and the former is the case for most applications.

On the other hand, if the standard library allocation routines use the `brk` heap, then this implementation could cause applications which use their own dynamic allocation package to malfunction. The problem is that `brk/sbrk` imply that the entire heap area above the returned pointer is available. Therefore, if the original thread uses a memory allocation package built on `brk/sbrk`, and the speculating thread uses a standard library allocation package built on `brk/sbrk`, it may be possible (depending on how each allocation package is implemented) for the original and speculating threads to simultaneously believe that they own the same memory region. This mixup could occur because the speculating thread would not be updating the data structures of the allocation package used by the original thread (and could be avoided if there were a way to ensure that the speculating thread uses the same memory allocation package as original normal execution).

Finally, my implementation of the memory allocation support package also limits the total amount of memory that the speculating thread can have allocated at any time.

6.3.2.3 General restrictions

The SpecHint design restricts the behavior of support routines in various ways in order to provide the guarantees discussed in Section 6.2. This section describes these restrictions and how they combine with the guarantees from previous sections.

The SpecHint design specifies that the behavior of all support routines (and any other work added to normal execution) be restricted such that an executing thread cannot trigger any exceptions, and can issue only system calls specified explicitly by the design, and guaranteed to return to user-level code. The design also specifies that the behavior of all support routines for speculating threads be additionally restricted such that, while speculating threads execute any such routine:

- They can modify only global data added by the binary modification tool, the speculating thread's stack, and pages mapped via a memory allocation support routine.
- They can leave the routine only by returning normally (i.e. returning to the return address provided when the routine was called), issuing system calls, calling a routine in shadow code or another of the support routines for speculating threads (except the thread init support routine), or jumping to a valid target address (as specified in Section 6.3.1.4) in shadow code. And,
- If they leave the routine to enter shadow code, they ensure that their stack pointer will contain a value in the safe stack area (as specified in Section 6.3.1.3) upon entering shadow code.

Combined with the guarantees discussed in Sections 6.3.1.3, 6.3.1.4, and 6.3.2.1, these are sufficient to ensure that Figure 6.4 captures the control flow of the speculating thread. Notice that the speculating thread can execute only support routines for speculating threads, the exception handling support routine, and shadow code. Therefore, since the only system calls that the speculating thread can issue are the call to set its signal mask such that it will block all but exception signals (from the thread init support routine), the calls to map and unmap memory from the memory allocation support package (which cannot produce direct or indirect output for the reasons discussed in the previous section), I/O prefetching calls (which the design assume cannot produce direct or indirect output), and the calls from shadow code (which cannot produce direct or indirect output for the reasons discussed in Section 6.3.1.2), the speculating thread cannot produce direct or indirect output by issuing system calls, cannot map devices into its address space, and will block all but exception signals. Moreover, since the speculating thread can only modify global data added by the binary modification tool, its stack, and pages mapped via a memory allocation support routine (because of copy-on-write checks, restrictions on system calls, and the specifications

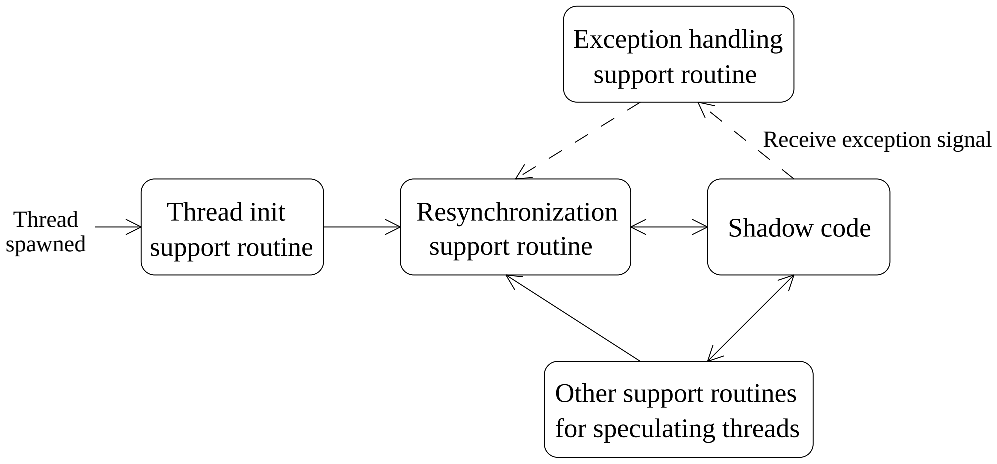
Exception handling support routine
Receive exception signal
Thread spawned
Thread init support routine
Resynchronization support routine
Shadow code
Other support routines for speculating threads

Figure 6.4: Allowed control flow of the speculating thread.

above), and since these cannot overlap pages mapped for original execution (as discussed in the previous section), the speculating thread will not modify data on any page mapped for original execution. By the same reasoning, the speculating thread also cannot modify code added by the binary modification tool. Finally, since the speculating thread will always have enough stack space to handle any signals it receives (as discussed in Section 6.3.1.3), and it cannot produce direct or indirect output by handling signals (as discussed in Section 6.3.2.1), it cannot produce direct or indirect output by triggering exceptions. Therefore, the design provides the set of guarantees discussed in Section 6.2 as being sufficient to ensure that the speculating thread will not produce direct or indirect output.

Finally, the design specifies that the behavior of all added normal execution be additionally restricted such that each sequence of added normal execution that an original thread performs (including any support routines it calls):

- Begins by changing its signal mask to block all signals and ends by changing its signal mask back to its prior setting.
- Allows it to modify only global data added by the binary modification tool, its stack, and pages mapped via a memory allocation support routine.
- Always completes. And,
- Upon completing, always leaves its stack and register values as they were when the sequence began, except for any dead values, and that the program counter should now specify the instruction which would have been executed next had the sequence not taken place.

Since the only system calls added normal execution can issue are the calls associated with spawning the speculating thread (which will not produce direct or indirect output for the reasons discussed above), and the calls associated with initializing the signal handling (which will not produce direct or indirect output for the reasons discussed in Section 6.3.2.1), this is sufficient to ensure that added normal execution will not produce direct or indirect output by issuing system calls, map devices, or change (without restoring) thread state maintained by the operating system. Moreover, since global data added by the binary modification tool and pages mapped via a memory allocation support routine will not overlap pages mapped for original execution, this is sufficient to ensure that added normal execution will not change (without restoring) any live values on any pages mapped for original execution. Therefore, the design provides the set of guarantees discussed in Section 6.2 as being sufficient to ensure that added normal execution will not produce direct or indirect output.

The SpecHint implementation

The SpecHint implementation falls short of these specifications in two ways. First, added normal execution does not change the original thread's signal mask. I claim that allowing signals to be received during added normal execution rather than original normal execution will rarely change the output of normal execution because it could only do so if original normal execution designated a signal handler whose effect would differ depending on what code the original normal execution was executing when the signal arrived. Second, the support routines call some byte and string manipulation routines in the standard C library, assuming they will behave as expected (e.g. that, if all strings are terminated and all specified buffers are properly accessible, they will return without triggering any exceptions after changing at most register values and the contents of any specified input buffers). The routines are: $bcopy$, $bzero$, $strcmp$, $strcpy$, and $strlen$. If these routines should not be trusted, they could easily be re-implemented within the support routines.

6.4 Summary

Performing speculative execution could change the behavior of a system in a variety of ways. While some potential changes would probably be considered beneficial (e.g. reducing I/O stall times), others might be perceived as malfunctions. A design for adding speculative execution should strive to avoid causing changes that would be perceived as malfunctions.

Every design will be based on some assumptions about what changes would be "safe". The *safety* of a design can be thought of as the likelihood that these assumptions are true in practice. This chapter begins by assuming that all changes in execution output are unsafe, except those that result from changes in allotment of physical and operating system resources. This may be sufficient for a design that is allowed to include operating system modifications. Otherwise, however, additional assumptions may be necessary.

The chapter proposes a set of additional assumptions that hold for a broad range of applications, and simplify the task of developing a design that can be both safe and effective for such applications. For example, one assumption is that original normal executions will access only the pages in their address space that contain their code, global data, stack(s) and heap(s). This assumption is based on the fact that, ordinarily, accessing any other page would cause the execution to trigger an exception. Relying on this assumption enables a design to allow a speculative execution sharing an address space with a normal execution to modify its own set of pages without fear that, in doing so, it will affect the behavior of the normal execution.

The chapter also describes how the SpecHint design guarantees that it will rely only on these and three additional assumptions that are also true for broad range of applications. The basic approach is to limit what output-producing actions speculative execution could possibly perform by limiting what code it could execute, and ensuring certain properties about that code.

Chapter 7

Experimental setup

My implementation of the SpecHint design consists of the *SpecHint tool* and the *SpecHint object files*. It takes advantage of the TIP informed prefetching and caching manager, and a prefetch-aware software striper [43]. In particular, speculating threads will issue *hint calls* to TIP, specifying what data to prefetch, and rely on TIP and the striper to schedule actual prefetches. In my evaluation, I compare the performance of speculating applications (produced using the SpecHint tool and object files) with unmodified applications, and applications that were manually modified to issue hint calls to TIP. The manual modifications were performed by the TIP research team. To provide insight into how much effort they expended in manually modifying these applications, I also describe the modifications they made.

Section 7.1 describes my evaluation environment. Section 7.2 describes details about the SpecHint tool and object files, and how the application executables used in the evaluation were produced. Finally, Section 7.3 describes the benchmarks used in my evaluation.

7.1 Evaluation environment

My experiments were conducted on an AlphaStation 255 (233 MHz Alpha 21064 processor) with 256MB of main memory. This machine has a fast, wide, differential SCSI adapter that hosts four HP2247 1 GB disks (15 ms average access time), so an average access time is equivalent to 3.5 million processing cycles. While this system is now quite out-of-date, I performed simulation experiments that project performance for current and future systems. The results of these experiments (discussed in Section 8.3) suggest that the performance benefit of of my approach will not decrease, and may even increase, as the gap between processing speeds and disk access times continues to widen.

My test machine runs Digital UNIX 3.2g with the unified buffer cache (UBC) module replaced by the *TIP informed prefetching and caching manager*[43, 41]. As described in Section 2.2.3.1, TIP was designed to optimize usage of the memory buffers allotted to the file cache when provided with disclosure hints that specify the future data accesses of an executing application. Thus, TIP simplifies SpecHint by allowing speculating threads

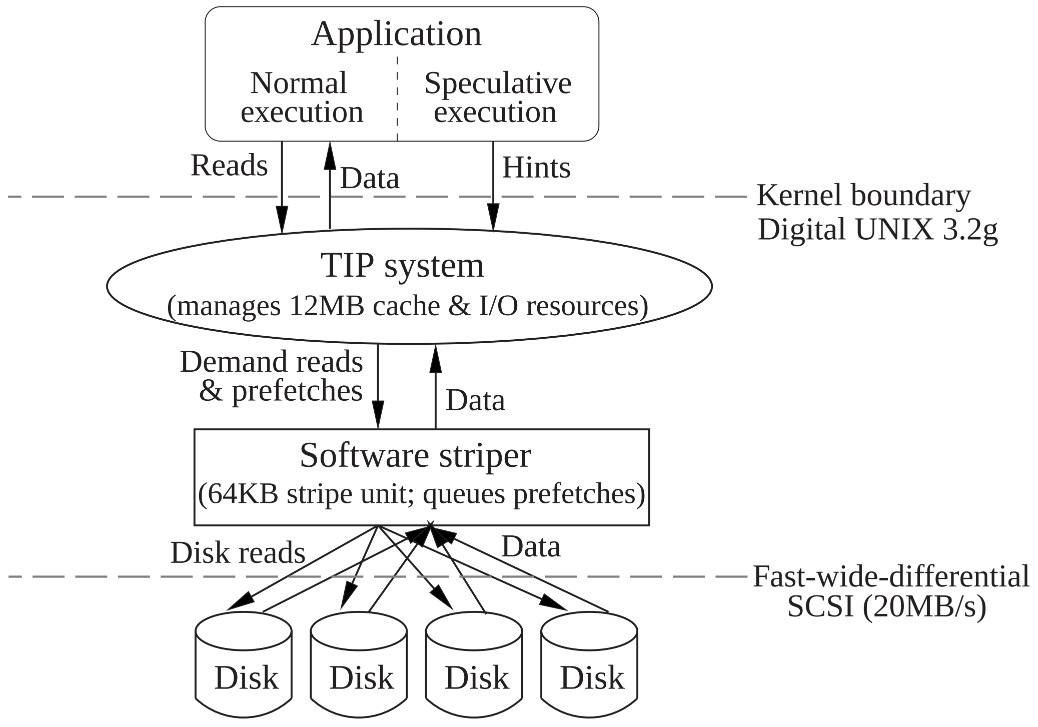

Figure 7.1: Interaction between speculating executable, TIP, software striper and disks.

to issue hint calls at any time (i.e. as quickly as they are able) and simply rely on TIP to schedule prefetches in a dynamic resource-aware manner. In my experiments, the file cache size was fixed at 12MB, the size used by the creators of the benchmark suite [43]. Current file caches are often larger, but the data sets used by the benchmarks have not been updated to reflect the growth in data set sizes since the benchmark suite was assembled. The file cache block size is 8 KB.

The disks are bound into a RAID Level 0 (i.e. no redundancy) disk array by a prefetch-aware software striper with a stripe unit of 64KB. This striper distinguishes between demand and prefech requests, and attempts to limit delay of demand requests by queueing prefetch requests as described in Section 7.1.1. Figure 7.1 illustrates the interactions between a speculating executable, the TIP prefetching and caching manager, the software striper and the I/O system.

Digital UNIX 3.2g is optimized for I/O in two ways. First, to increase the disk positioning efficiency and decrease the CPU cost of sequential disk accesses, it will cluster up to eight contiguous block accesses (i.e. 64 KB) into a single disk request. On the testbed system, block contiguity checks take the 64 KB stripe unit into account to ensure that each cluster results in a request to a single disk. Second, to decrease the latency of sequential file accesses, sequential readahead is initiated for the same number of blocks as the current run of sequential accesses to a file, up to a maximum of eight clusters of eight blocks each. For example, if an application sequentially reads four blocks in a file, the system will initiate readahead for the next four blocks in the file. With TIP, hinted accesses do not trigger the readahead policy. Therefore, an application that provides accurate hints can avoid unnecessary readahead.

The executables and data files used in the experiments were written to an empty file system. Therefore, files are more likely to be laid out sequentially, and files in the same directory are more likely to be proximate, than in a mature file system (which would be fragmented due to deletions and insertions over time). The file cache is flushed before each run. All reported results are averages over five runs.

7.1.1 Prefetch-aware software striper

The software striper [43, 41] makes multiple disks appear as a single disk to the rest of the system, enabling I/O parallelism within a single file system. The file system and TIP are both aware of the 64 KB stripe unit such that they will not issue requests that span multiple disks. When the striper receives an I/O request, it maps the request to the appropriate disk and disk blocks. Then, if the request is a demand request, it immediately forwards the request to the appropriate disk driver. If the request is a prefetch request, it either forwards the request or queues the request internally.

The striper queues prefetch requests internally in order to reduce the amount by which prefetch requests (including readahead requests) will delay demand requests. It maintains a prefetch queue for each disk and, to reduce disk positioning times, sorts the requests in each queue according to the CSCAN (i.e. circular scan) scheduling algorithm [14]. In my experiments, unless otherwise specified, the striper will issue prefetch requests to a disk only when the disk has less than two (prefetch and/or demand) requests outstanding. The striper is allowed to issue a prefetch request to a disk that already has one request outstanding in order to eliminate disk idle time between requests.

The implementation of the striper has two shortcomings that could potentially hurt the performance of speculating applications. First, the striper does notice if a prefetch request in one of the striper's prefetch queues is promoted to a demand request (by a demand access). Therefore, a promoted request could potentially sit in one of those queues for a substantial amount of time. Second, once a prefetch request has been received by the striper, there is no way to cancel it. Therefore, even if it is determined that a prefetch will prove to be useless while it is in one of the striper's prefetch queues, the prefetch will still occur, possibly delaying demand requests or other prefetch requests. These do not affect the performance of the manually modified applications used in my evaluation since those applications generate few or no incorrect hints. However, it may hurt the performance of the speculating applications that generate many incorrect hints. These effects were not measured in this dissertation work.

7.2 SpecHint tool and object files

The SpecHint tool and object files are my implementation of the SpecHint design. The tool transforms Digital Unix 3.2 (Alpha architecture) binaries, and is implemented in 19,000 lines of sparsely commented C code. The object files, which will be linked into a speculating executable, mainly consist of the code for the speculative execution support routines and data structures. They were produced using the native *cc* compiler for Digital Unix 3.2g from 6,000 lines of heavily commented assembly code.

To simplify the implementation of the SpecHint tool, the input binaries must be statically linked and, unlike executables, retain their relocation information. Projects like EEL [29], ATOM/OM [54] and Etch [45] have demonstrated that executables can be modified safely. To handle shared libraries, the tool would need to be able to produce modified versions of shared libraries for use with speculating executables, and cause the speculating thread to abort its current speculation if it attempts to call a routine in a shared library for which it is unable to find a modified version. This should not present any technical difficulties.

The SpecHint tool is relatively unsophisticated in terms of static optimizations. For example, it does not perform any loop optimizations to reduce copy-on-write checks, even though such optimizations may be able to substantially decrease the dilation factor (the relative speed of speculative execution to normal execution). (In terms of reducing copy-on-write checks, the tool implements only the optimization explained in Section 6.3.1.1 for eliminating copy-on-write checks before loads and stores off the stack pointer and global pointer.)

The shadow code versions of some lock handling routines for thread synchronization are specified as dummy routines. It is unnecessary and counter-productive to synchronize data accesses by the original and speculating threads since the speculating thread cannot change data values seen by the original thread, and needs to execute ahead of the original thread to produce useful hints.

7.2.1 Producing executables

Each of the benchmark applications used in my evaluation has two versions of source code, the original source code and source code manually modified to issue TIP hints. I generated object files from this source code using the native `cc` compiler for Digital Unix 3.2g with the `-O2` optimization flag. For linking, I used the standard linker for Digital Unix 3.2g. During my evaluation, I used several executable versions for each application. The `Original` version of each application is the executable produced by linking the object files generated from the original application source code. The `Manual hints` version of each application is the executable produced by linking the object files generated from the application source code manually modified to issue hints. The evaluation also use several different speculating versions of each application. Figure 7.2 illustrates the process I used to produce these speculating versions. First, I produced a binary by linking the object files generated from the original application source code, with the SpecHint object files and

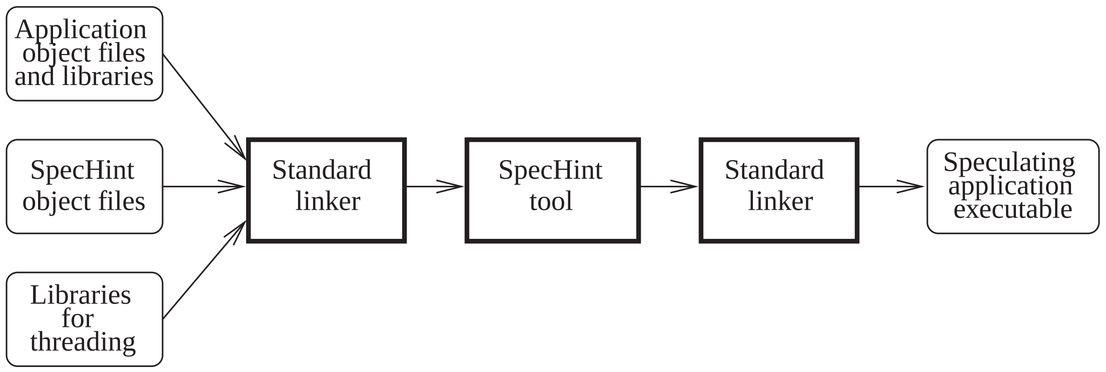

Figure 7.2: Producing speculating executables. The application object files shown in the figure are produced from original application source code. The SpecHint object files contain the SpecHint support routines. The libraries to support threading are required because the original application is single-threaded.

the standard libraries that provide thread support (the reentrant C library, the `mach` library and the POSIX-compliant `pthreads` library for Digital Unix 3.2g). I used the `-r` linker flag, which generates (non-executable) binaries that still retain relocation information (required by the SpecHint tool). Next, I passed this resulting binary as input to the SpecHint tool, which generates a modified binary. Finally, I produced an executable by passing this modified binary to the linker, without any flags (so that the linker strips out relocation information). For comparability, all application executables used in the evaluation are linked statically.

I produced multiple speculative versions of each benchmark application. *Naive* refers to the version of each benchmark application that was transformed according to my implementation of the SpecHint design. My implementation can also, optionally, produce transformed applications that include additional mechanisms, not specified by the SpecHint design. These optional mechanisms (described in Sections 5.2.3, 4.2.4, and 4.2.5) attempt to improve the effectiveness of speculative execution. I produced versions of each benchmark application that include various combinations of these mechanisms in order to isolate their cost and benefit. In particular, version names which include *Filter* incorporate the mechanism described in Section 5.2.3 for predicting which hints are likely to be incorrect and filtering out those hints. Version names which include *Clear*, *Set*, or *Int* incorporate the simple heuristic described in Section 4.2.5 that attempts to improve the effectiveness of speculative execution by choosing specific values for stale memory locations. Specifically, *Clear* executables logically clear the bits in stale memory locations (i.e. set them to zero), *Set* executables logically set the bits in stale memory locations (i.e. set them to an integer value of -1), and *Int* executables logically fill stale memory locations with the integer value of one. Finally, version names that include *Slicing* incorporate the mechanism described in Section 4.2.4 which enables speculating threads to detect and skip code that they do not need to execute to generate correct hints.

Benchmark	Application	Description
Agrep	Text search utility	Search for a string in many small files
XDataSlice	Data visualization tool	Retrieve slices of a three-dimensional data set
Gnuld	Link editor	Generate an executable from many object files
Postgres 80%	Relational database	Perform a database join (finds many matches)
Postgres 20%	Relational database	Perform a database join (finds fewer matches)
Sphinx	Speech recognition tool	Recognize a recording

Table 7.1: Summary of benchmarks. These benchmarks come from the TIP benchmark suite [43, 41]. The TIP benchmark suite also includes a `Davidson` benchmark which I was unable to use because that application is written in Fortran.

7.3 Benchmark applications

I evaluated SpecHint using the six TIP benchmarks [43, 41] listed in Table 7.1. The TIP benchmark suite consists of seven benchmarks using six applications, but I was unable to include one of them because the application (*Davidson*) was implemented in Fortran and my SpecHint tool contains some shortcuts such that it may refuse to modify a binary not produced by the native `cc` compiler. There are two major advantages to using this benchmark suite. First, it is one of the few existing I/O benchmark suites that relies on a variety of real-world, non-scientific applications. Second, the creators of the benchmark suite also produced a version of each application's source code that has been manually modified to initiate prefetching by issuing TIP hints (which I used without change).

7.3.1 Agrep

The *Agrep* benchmark uses version 2.04 of the Agrep application [71]. Agrep, a variant of the standard UNIX Grep utility, is a fast full-text pattern matching utility that allows matching errors. The benchmark searches a bunch of text files for a simple string that does not occur in any of the files. In terms of data requests, it loops through the files specified in its command line, opening and sequentially reading each file entirely. That is, the data requests issued by the benchmark are completely determined by its command line. The data files are 1349 source files for the Digital Unix kernel. They occupy a total of 2922 blocks and have sizes distributed as shown in Table 7.2.

Manually modifying Agrep to issue TIP hints is easy. Before reading any files, Agrep loops through all of the filenames specified in its command line to check whether each file exists. Manually modifying Agrep simply involves inlining a TIP hint call after a successful check as shown in Figure 7.3.

	Number of 8 KB blocks in the file									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10-38**
Number of files	798	279	98	52	37	19	16	9	9	32
Cumulative % files	59%	80%	87%	91%	94%	95%	96%	97%	98%	100%
Cumulative % blocks	27%	46%	56%	64%	70%	74%	78%	80%	83%	100%

Table 7.2: Distribution of data file sizes for Agrep benchmark. Sequential readahead by the operating system is completely ineffective for files that fit in a single cache block, and is most effective for long sequential accesses.

Code with inlined hint call

```
while (another_unchecked_filename) {
  if (stat(unchecked_filename, &buf) == 0) {
    whole_file_hint(unchecked_filename);
    add_to_list_of_files_to_search(unchecked_filename);
  }
}
```

Figure 7.3: Pseudo-code showing how Agrep can be modified to issue effective TIP hints simply by inlining a TIP hint call.

7.3.2 XDataSlice

The *XDataSlice* benchmark uses a modification of version 2.2 of the XDataSlice application [11]. XDataSlice is a data visualization tool developed by the National Center for Supercomputing Applications (NCSA) that, among other things, allows users to view false-color representations of arbitrary planar slices through a three-dimensional data set. The NCSA version of the application was limited to data sets that fit in memory. The version used as the original in my benchmarks was modified to load data dynamically from larger data sets [41]. The benchmark retrieves a fixed set of 25 random slices through a 512 MB three-dimensional data set. In terms of data requests, it loops through the entries in an input file specified in its command line and, for each entry, retrieves the data for the slice specified by that entry. Because each slice is an arbitrary cut through a three-dimensional data set, the accesses to the 512 MB data file are strided, but cannot be captured with a single stride size.

Manually modifying XDataSlice to issue TIP hints was complicated by the fact that XDataSlice issues data requests via a two-layer library (that NCSA developed for accessing files in a format called the Hierarchical Data Format). For each slice, the XDataSlice-specific code issues a request for the corresponding list of "blocks" in the data set to the higher `DFSD` layer of the library. The `DFSD` layer loops through this list of blocks, calculating and issuing the corresponding request for logical byte ranges in objects to the lower `H` layer of the library. For each such request, the `H` layer calculates and issues the corresponding request for actual byte ranges in a file. Therefore, only the `H` layer has enough

Original code

```
foreach dataset_block {
  logical_addrs = get_logical_addrs(dataset_block);
  H.request(logical_addrs);
}
```

Modified loop-split code

```
i = 0;
foreach dataset_block {
  logical_addrs[i] = get_logical_addrs(dataset_block);
  H.hint(logical_addrs[i]);
  i++;
}
i = 0;
foreach dataset_block {
  H.request(logical_addrs[i]);
  i++;
}
```

Figure 7.4: Pseudo-code showing how the loop in the DFSD layer of the XDataSlice application is transformed such that TIP hints will be issued effectively.

information to issue a TIP hint. However, the `H` layer cannot determine what data will be needed in the future. Since violating application modularity is poor programming practice, this layering complicated the process of manually modifying XDataSlice to issue effective TIP hints. To preserve the modularity of the application, the manually modified XDataSlice extends the interface to the `H` layer by a call whose arguments specify logical byte ranges that will be needed in the future, and that issues corresponding TIP hint calls. In addition, as shown in Figure 7.4, the loop in the `DFSD` layer was split after promoting a scalar loop variable to an array variable such that these hint calls can be issued early enough to hide I/O latency without requiring all work to be repeated. This type of transformation, commonly called *(inner) loop distribution*, is commonly performed in the context of array-based codes to increase parallelism.

7.3.3 Gnuld

The *Gnuld* benchmark uses version 2.04 of the Gnuld application. Gnuld is the Free Software Foundation's object code linker. The benchmark creates an executable by linking a bunch of object files. In terms of data requests, it performs multiple passes over the files specified in its command line, reading different portions of each file on each pass. During the first pass, for each file, it reads some headers from the beginning of a file, and then some symbol information from an offset that is obtained from those headers. Subsequent passes read different object file sections, where the offset of a particular section in a particular file is obtained from the initial headers. The data files are 562 object files for the Digital Unix

Original code

```
for (i=0; i < NUM_OFILES; i++) {
  read(ofile[i], ofile[i].headers, header_size);
  lseek(ofile[i], SEEK_SET, ofile[i].headers.offset_of_symbol_info);
  read(ofile[i], ofile[i].symbol_info, symbol_info_size);
}
```

Modified loop-split code

```
for (i=0; i < NUM_OFILES; i++)
  hint(ofile[i], 0, header_size);
for (i=0; i < NUM_OFILES; i++) {
  read(ofile[i], ofile[i].headers, header_size);
  hint(ofile[i], ofile[i].headers.offset_of_symbol_info, symbol_info_size);
}
for (i=0; i < NUM_OFILES; i++) {
  lseek(ofile[i], SEEK_SET, ofile[i].headers.offset_of_symbol_info);
  read(ofile[i], ofile[i].symbol_info, symbol_info_size);
}
```

Figure 7.5: Pseudo-code showing how the loop for the initial pass in Gnuld is transformed. A new loop is inlined before the original loop, and the original loop is split such that TIP hints can be issued early enough to be effective.

kernel, occupying a total of 7361 blocks.

Manually modifying Gnuld to issue TIP hints involved splitting one loop, and inlining three new loops. First, a new loop was inlined before the loop that implements the first pass to hint the headers at the beginning of each file. Next, the loop that implements the first pass was split to enable effective hinting of the read requests for symbol information. Pseudo-code for these transformations is shown in Figure 7.5. Finally, new loops were added before the two loops that implement the subsequent passes in order to generate hints for the reads in those loops.

7.3.4 Postgres

The *Postgres 20%* and *Postgres 80%* benchmarks use a modification of version 4.2 of the Postgres application [57]. Postgres is an object-oriented relational database developed at the University of California at Berkeley. The version used as the original in the benchmark has better performance than the Berkeley version because it was modified to more fully exploit caching using a technique that is employed by most commercial databases [43, 41]. In particular, the Berkeley version performs a join of two relations by looping through the tuples of the outer relation once. During this loop, for each outer tuple, it checks the inner relation's index for a matching inner tuple and, upon finding such a match, reads the tuple and outputs the match. The version used as the original in the benchmark instead performs a join of two relations by looping through the tuples of the outer relation twice. During the first loop, for each outer tuple, it checks the inner relation's index for a matching inner tuple and, if one exists, simply records the location of that tuple. Then, during the second loop, for each outer tuple, if there is a recorded match, it reads the recorded inner tuple and output the match. This improves performance because, while it is being used during the first pass, the inner tuple's index does not need to compete for cache space with the inner tuples.

Each of the benchmarks perform a join of two relations. In terms of data requests, the outer tuple accesses are sequential, but the index accesses and the inner tuple acccesses look random from the operating system's perspective. In the *Postgres 20%* (*80%*) benchmark, 20% (80%) of the outer relation tuples have a match in the inner relation. There are 20,000 tuples in the 3.2 MB unindexed outer relation, and 200,000 tuples in the 32 MB indexed inner relation. The index of the inner relation is 5 MB.

The manually modified version of Postgres issues hints for the first sequential read of the outer relation before executing the first loop. It also issues hints for all the matching inner tuples found during the first loop before executing the second loop. Postgres could be (but was not) manually modified to also provide hints for the second sequential read of the outer relation. This modification was not performed because it would have required additional effort without providing much benefit. In particular, since TIP maintains a single, ordered hint queue for each process, it would have required keeping track of the interleaving between reads of the outer relation and reads of matching inner tuples. This additional effort would not have provided much benefit since operating system sequential file readahead should automate prefetching of the outer relation. Postgres could also be (but was not) manually modified to provide hints for accesses to the index. This modification was not performed because the structure of the code makes it difficult to achieve (the index is a B-tree and an index lookup is a single recursive call). Finally, manually modifying Postgres is complicated by the fact that it maintains its own 100 block cache. Since some accesses to the inner relation can and do hit in this cache, in order to avoid TIP's hint queue head-of-line blocking behavior (described in Section 2.2.3.1), the manually modified Postgres needs to (and does) issue a hint cancellation call whenever such a cache hit occurs.

7.3.5 Sphinx

The *Sphinx* benchmark uses a modification of version 8 of the Sphinx application [30]. Sphinx is a speaker-independent, continuous-voice speech recognition system developed at Carnegie Mellon University (CMU). The performance of the CMU application degrades drastically when there is insufficient main memory for it to execute mostly in-core. The version used as the original in the benchmark was modified to load language model data from disk as needed. This enabled it to run almost as quickly, with the same data, on a machine with much less memory [43, 41]. The benchmark is to recognize an 18-second recording commonly used in Sphinx regression testing. There are two phases to the execution: an initialization phase and a recognition phase. In terms of data requests, it sequentially reads the phone, distribution map, dictionary, and senone probability distribution files entirely, as well as reading part of the language model file. During the recognition phase, it performs several pruning passes during which it requests data as necessary from the language model file, which mostly consists of a table of conditional probabilities of word-pairs and word-triples. From the operating system's perspective, these accesses look random. The four sequentially read files total about 15 MB, while the language model file is almost 176 MB.

The manually modified version of Sphinx hints accesses to the table of conditional probabilities in the language model file, but the timing of these hints is limited by the fact that the recording is broken into 10 ms acoustical frames and recognition proceeds on a frame-by-frame basis (i.e. the results of the last frame determine the requests that will be issued on behalf of the next frame). Like Postgres, Sphinx maintains its own cache. To avoid the hint queue head-of-line blocking issue, the manually modified Sphinx keeps track of the hints it issues, checks them against the reads it issues, and cancels hints as necessary. Manually modifying Sphinx to issue hints involved adding a few support routines and data structures, and inlining code in nine different procedures, although this was not strictly necessary in four cases.

7.4 Summary

The SpecHint design assumes operating system support for I/O prefetching. In particular, it assumes that the operating system will manage memory and I/O resources such that processes can issue prefetch calls without worrying about about how to schedule prefetches such that they will hide disk latency but not hurt memory or I/O performance. This support simplifies the design by enabling speculating threads to simply issue prefetch calls as quickly as they can identify what data to prefetch. The SpecHint implementation relies on the TIP informed prefetching and caching manager, and a prefetch-aware software striper, to provide this support.

The evaluation uses six benchmarks from the TIP benchmark suite. The benchmark applications are all real-world I/O-intensive applications, and include a wide variety of access patterns and application types. Two versions of each application's source code were used to produce the executables used in the evaluation. The original version of the source code, which does not contain hint calls, was used to produce both the base-line non-hinting application executable, and the speculating executables. The manual hints application executable was produced using source code that was manually modified to issue prefetching hints. For many of the benchmark applications, a substantial amount of work involved in manually modifying the benchmark applications to issue hints.

Chapter 8

Evaluation

This chapter presents an evaluation of the SpecHint design and implementation. The evaluation has three main directives: 1) it demonstrates how successfully this implementation of the speculative execution approach achieves its overall goal of reducing application elapsed times; 2) it investigates what limits SpecHint's ability to be more effective; and 3) it examines the performance impact of key elements of the SpecHint design, validating performance claims made in earlier chapters and demonstrating the strengths and weaknesses of this design.

In order to demonstrate the potential of the SpecHint design and implementation, the chapter begins by examining the performance of speculating applications in tests during which the tested application is the only application executing on the testbed. First, Section 8.1 extensively examines the performance of my implementation of the SpecHint design, which I refer to as *naive SpecHint*. Next, Section 8.2 examines the cost and benefit of several optional mechanisms in the SpecHint implementation that attempt to improve the effectiveness of speculative execution. Section 8.3 then projects how the trends in processor and disk speeds will affect the benefit that this approach will be able to deliver in the future.

Since speculative execution is mostly confined to consuming spare processing cycles, one question with this approach is whether it can still provide benefit when there is contention for the processor. To answer this question, Section 8.4 analyzes performance when multiple applications are executed concurrently.

Finally, the SpecHint design requires that applications undergo a binary transformation step (described in Section 7.2) before they can benefit from the speculative execution approach. Section 8.5 presents the cost of this transformation step in order to demonstrate that it is not prohibitively large.

8.1 Single application, naive SpecHint

To evaluate the performance of the SpecHint design, I compare the performance of *naive SpecHint executables* (executables produced using the SpecHint tool, according to the

SpecHint design) to the performance of both *original, non-hinting executables* (produced directly from each benchmark application's original source code) and *manual hinting executables* (produced from source code that was manually modified to include TIP hint calls). Comparing against original, non-hinting executables reveals the performance benefit that naive SpecHint provides automatically. Comparing against manual hinting executables essentially reveals how close naive SpecHint's performance is to the optimal performance that could be obtained through application-level I/O prefetching. I begin by discussing overall results. Subsequent sections examine the behavior of naive SpecHint in greater detail.

8.1.1 Overall performance

Figure 8.1 shows the elapsed times of naive SpecHint executables and manual hinting executables relative to original, non-hinting executables. These graphs show that performing speculative execution can substantially reduce the elapsed times of all but one of the benchmarks (Sphinx), especially when the data is spread across multiple disks.

The graphs also show that the performance of the naive SpecHint executable is comparable to the performance of the manual hinting executable for two of the six benchmarks, Agrep and XDataSlice. The main reason that SpecHint delivers near-optimal performance for these benchmarks is that, during these benchmarks only, the data unavailable during speculative execution seldom if ever impedes speculative execution's ability to identify future data requests. The benchmark-specific dependencies of speculative execution on unavailable data values turn out to be the main determinant of the performance of speculative execution for a benchmark. Later sections show that the speed at which speculative execution is able to identify future data requests is also a concern for some of the benchmarks.

The geometric mean elapsed time of the naive SpecHint executables, as a fraction of the elapsed time of the original, non-hinting executables, is 0.91, 0.68, and 0.55 for 1, 2, and 4 disks, respectively. The geometric mean elapsed time of the manual hinting executables, as a fraction of the elapsed time of the original non-hinting executables, is 0.79, 0.53 and 0.39 for 1, 2 and 4 disks, respectively. That is, both speculating and manual hinting executables provide much less benefit when the data is spread over fewer disks, but manual hinting executables are less sensitive than speculating executables to a lack of I/O parallelism. The former is not surprising since fewer disks means less I/O bandwidth, and prefetching requires spare I/O bandwidth to improve performance. The latter is primarily because speculating executables are often unable to issue hints as early as manual hinting executables, and issuing hints later is more likely to hide less I/O latency on a system with less spare I/O bandwidth. In particular, multiple disks enable prefetching to be overlapped with servicing of demand requests so that, to hide the same amount of disk latency, prefetches can be issued much later on a multi-disk system than they would need to be issued on a single disk system. A secondary reason is that speculating executables are more likely to issue incorrect hints, and a useless prefetch issued as the result of an incorrect hint is more likely to hurt performance on a system will less spare I/O bandwidth. In particular, on a system with more spare I/O bandwidth, prefetches are more likely to consume

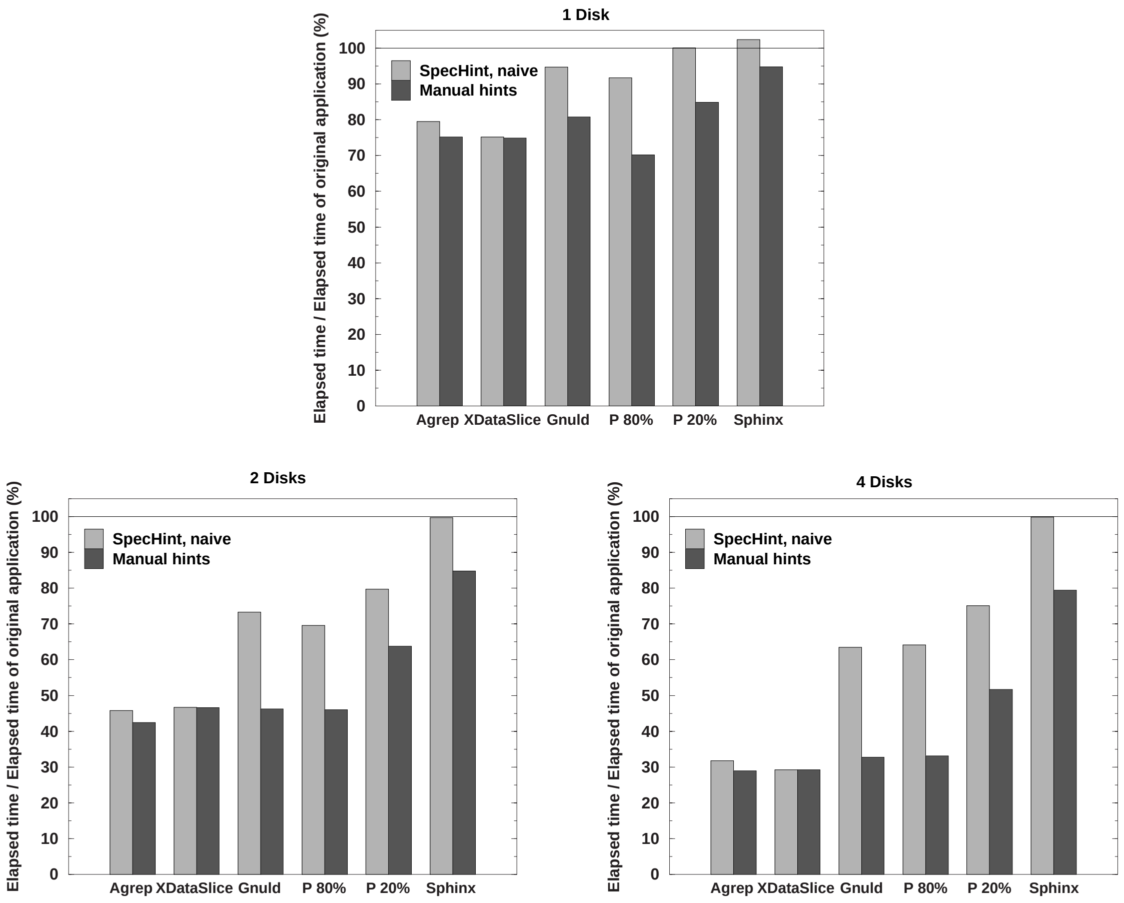

Figure 8.1: For each of the benchmarks summarized in Table 7.1, these graphs show the elapsed time of the naive SpecHint executable (the executable produced using my implementation of the SpecHint design) and the manual hinting executable (the executable produced from source code manually modified to include prefetch hint calls) relative to the original, non-hinting executable when the file system is striped across one, two, or four disks.

bandwidth that would otherwise have been unused, so useless prefetches are less likely to inadvertently delay demand fetches.

Table 8.1 gives the raw numbers used to generate the geometric means and the graphs of Figure 8.1. The percentage of an original executable's elapsed time during which the original executable is stalled on I/O places an upper limit on the maximum benefit that can be obtained by hiding I/O latency (e.g. via prefetching). The original executables for all of the benchmarks except Sphinx spend at least 70% of their elapsed times stalled on I/O. Sphinx spends less than 40% of its time stalled on I/O. Notice that the original, non-hinting executables derive little if any benefit from multiple disks. This is because they are single-threaded and issue synchronous data requests, and therefore have at most one I/O request

Benchmark	Version	1 Disk		2 Disks		4 Disks	
		Elapsed	Idle	Elapsed	Idle	Elapsed	Idle
Agrep	Original	23.4	21.3	23.6	21.5	21.1	19.0
		(0.80)	(0.80)	(1.10)	(1.10)	(0.70)	(0.60)
	Naive	18.6	0.5	10.8	0.4	6.7	0.5
		(0.00)	(0.00)	(0.50)	(0.00)	(0.40)	(0.20)
	Manual	17.6	15.6	10.0	8.0	6.1	4.1
		(0.00)	(0.00)	(0.40)	(0.40)	(0.30)	(0.30)
XDataSlice	Original	301.4	261.0	303.5	263.9	332.4	292.1
		(0.30)	(0.50)	(2.30)	(2.20)	(2.30)	(2.10)
	Naive	226.7	2.2	141.7	2.0	97.2	2.0
		(0.20)	(0.10)	(0.50)	(0.20)	(0.60)	(0.20)
	Manual	225.6	194.5	141.5	110.6	96.9	65.9
		(0.30)	(0.30)	(1.00)	(1.20)	(0.30)	(0.30)
Gnuld	Original	97.0	86.7	104.9	95.0	109.4	99.4
		(1.20)	(0.90)	(0.40)	(0.30)	(0.70)	(0.50)
	Naive	91.9	5.7	76.9	5.4	69.5	5.6
		(0.90)	(0.30)	(0.50)	(0.30)	(0.70)	(0.10)
	Manual	78.4	68.8	48.5	39.0	35.8	26.3
		(0.90)	(0.90)	(0.70)	(0.70)	(0.50)	(0.40)
Postgres 80%	Original	231.4	186.7	224.7	178.7	231.6	185.9
		(1.00)	(0.90)	(3.50)	(2.00)	(3.50)	(2.40)
	Naive	212.1	5.1	156.3	3.3	148.5	2.8
		(1.30)	(0.30)	(2.90)	(0.30)	(1.70)	(0.10)
	Manual	162.4	117.6	103.4	59.0	76.7	31.5
		(6.50)	(6.50)	(2.30)	(2.40)	(1.60)	(1.50)
Postgres 20%	Original	86.5	62.2	84.9	60.2	86.3	61.4
		(0.90)	(0.30)	(1.10)	(0.20)	(0.90)	(0.30)
	Naive	86.6	4.0	67.7	2.5	64.8	2.4
		(0.50)	(0.10)	(0.30)	(0.10)	(0.60)	(0.10)
	Manual	73.4	48.6	54.2	30.0	44.6	19.9
		(0.50)	(0.70)	(1.30)	(0.90)	(0.50)	(0.40)
Sphinx	Original	236.2	84.2	238.9	87.0	229.3	77.2
		(1.10)	(0.50)	(1.70)	(0.70)	(1.10)	(1.00)
	Naive	241.8	2.3	238.2	1.7	229.1	1.5
		(0.90)	(0.10)	(1.20)	(0.10)	(2.40)	(0.10)
	Manual	223.9	66.5	202.5	46.9	182.0	27.5
		(2.60)	(0.80)	(0.80)	(0.30)	(0.50)	(0.40)

Table 8.1: Raw figures for the elapsed times and idle times of the original, naive SpecHint and manual hinting executables on one, two, and four disks. All times are given in seconds. The idle times for the original and manual hinting executables are the amount of time during which these executables were stalled on I/O. The idle times for the naive SpecHint executables are the time during which both the original thread and the speculating thread were blocked (e.g. while the original thread is blocked on a read call and the speculating thread is blocked on a page fault). The figures in parenthesis give the size of the 95% confidence intervals.

outstanding at any time. Furthermore, they tend to issue requests of such modest sizes that the requested data resides on a single disk.

8.1.2 Hinting performance

The sole purpose of performing speculative execution is to generate correct hints for the future data needs of the target normal execution. To evaluate the hinting performance of the SpecHint design, this section compares the ability of naive SpecHint executables to generate hints with that of the manual hinting executables.

Table 8.2 shows the number of read calls issued by each benchmark, and the percentage of these read calls, and the blocks specified by these read calls, which were correctly hinted by the naive SpecHint and manual hinting executables. It also shows the number of incorrect hints, and of incorrectly hinted blocks. Only one set of figures is given for the manual hinting executables because these executables issue the same hints every run, regardless of system parameters like the number of disks. One, two and four disk figures are given for the naive SpecHint executables because the hints generated by speculating executables depend on when and how many spare cycles are available for speculative execution.

In the benchmarks for which the naive SpecHint executables produce the same performance gains as the manual hinting executables – Agrep and XDataSlice – the naive SpecHint executables are as successful at hinting as the manual hinting executables. For all the other benchmarks, the naive SpecHint executables generate noticeably fewer correct hints than the manual hinting executables, and also generate many incorrect hints. This is not surprising since, for all of the applications except Agrep and XDataSlice, the values that are unavailable during speculative execution effect whether speculative execution is able to identify future data requests correctly. The manual hinting executables, on the other hand, generate incorrect hints only for Postgres. As described in Section 7.3, Postgres maintains an internal cache and does not issue read calls for data in that cache. Although the manually modified Postgres attempts to avoid issuing hints for data that will be in the internal cache, it will infrequently issue such a hint. Sphinx also maintains an internal cache, but the manually modified Sphinx is able to avoid issuing hints for data in that cache.

In general, a prefetching application will spend more time stalled on I/O on a system with fewer disks because I/O bandwidth will be more limited, so a speculating application will have more opportunity to produce hints on a system with fewer disks. This effect can be seen by comparing the total number of hints generated in the one disk and four disk configurations. The additional hints may not be useful, however; for example, on a single disk, the naive SpecHint executable mainly produces more incorrect hints for the Postgres benchmarks. Notice that, somewhat ironically, the more successfully a speculating application generates hints, the less opportunity it will have to generate more hints, because normal execution will spend less timed stalled on I/O.

The figures for the number of correctly hinted read calls show that even the manual hinting executables do not issue hints for all the read calls of every benchmark. In some cases, this is because it is unnecessary. For example, the Agrep benchmark issues a read

Benchmark	Read calls	Read call blocks
Agrep	4277	2928
XDataSlice	46356	46352
Gnuld	13037	20091
Postgres 80%	31245	31243
Postgres 20%	8678	8676
Sphinx	65282	77714

Benchmark	Version	Disks	Correctly hinted				Incorrectly hinted	
			Calls		Blocks		Calls	Blocks
Agrep	Naive	1	2915	(68%)	2915	(100%)	0	0
	Naive	2	2915	(68%)	2915	(100%)	0	0
	Naive	4	2910	(68%)	2910	(99%)	0	0
	Manual	–	2922	(68%)	2922	(100%)	0	0
XDataSlice	Naive	1	45241	(98%)	45241	(98%)	0	0
	Naive	2	45236	(98%)	45236	(98%)	0	0
	Naive	4	45246	(98%)	45246	(98%)	0	0
	Manual	–	45241	(98%)	45241	(98%)	0	0
Gnuld	Naive	1	8215	(63%)	14659	(73%)	2266	2605
	Naive	2	8215	(63%)	14634	(73%)	2249	2531
	Naive	4	8169	(63%)	14580	(73%)	2233	2346
	Manual	–	10225	(78%)	17273	(86%)	0	0
Postgres 80%	Naive	1	14531	(47%)	14531	(47%)	8813	8813
	Naive	2	14697	(47%)	14697	(47%)	4593	4593
	Naive	4	14155	(45%)	14155	(45%)	3935	3935
	Manual	–	16083	(51%)	16083	(51%)	242	242
Postgres 20%	Naive	1	3922	(45%)	3922	(45%)	2903	2903
	Naive	2	3771	(43%)	3771	(43%)	2386	2386
	Naive	4	3604	(42%)	3604	(42%)	2081	2081
	Manual	–	4384	(51%)	4384	(51%)	71	71
Sphinx	Naive	1	10288	(16%)	16006	(21%)	164	256
	Naive	2	7310	(11%)	11887	(15%)	1585	2319
	Naive	4	2582	(4%)	7205	(9%)	658	713
	Manual	–	62586	(96%)	74871	(96%)	0	0

Table 8.2: Hinting performance of the naive SpecHint and manual hinting executables. The top table shows, for each benchmark, the number of read calls, and the number of blocks specified by these read calls. The bottom table shows how many of these calls (and blocks) were correctly hinted, as well as the number of incorrect hint calls (and blocks). The hinting performance of the manual hinting executables does not vary with the number of disks. The original executables are not included in this table because they do not generate any hints.

call at the end of each of the 1347 data files. Since these end-of-file calls do not fetch any data, there is no point in issuing a hint for them. Both the naive SpecHint and manually modified Agrep do not issue hints for end-of-file calls, while issuing hints for almost all (>99%) of the benchmark's data-returning calls. As another example, for each data file, the Gnuld benchmark issues several small read calls in a row to read in fixed-size data structures at the beginning of the file (the object file headers) which fit in the first block of the file. The manually modified Gnuld recognizes that it is unnecessary to issue multiple sequential hints for the same block, so it issues a single hint for the first block of each file. (The SpecHint implementation does not contain logic to avoid unnecessary hints by detecting sequential accesses to the same block, but the cost of issuing such an unnecessary hint is negligible.) On the other hand, the manually modified Postgres does not issue hints for a subset of its read calls because modifying Postgres to issue effective hints for these calls would be very complicated, requiring many non-localizeable modifications. This provides an opportunity for SpecHint to improve on the performance of a manual hinting executables. Unfortunately, speculative execution also has difficulty issuing hints for those read calls. The situation with Sphinx is similar; the code would need to be restructured substantially to allow more read calls to be hinted effectively.

The difference in the elapsed times of the naive SpecHint and manually modified Gnuld and Postgres are disproportionately greater than the difference in the percentage of correctly hinted read calls. This is because the performance effect of a correct hint depends on how early it is generated. If a hint is not generated early enough, it will probably provide no performance benefit even if correct. Therefore, to understand the performance of the SpecHint design, it is also necessary to consider the prefetching performance of the executables.

8.1.3 Prefetching performance

The main benefit of the hints generated during speculative execution is that they can lead to more accurate prefetching than the operating system's readahead policy. (Hints may also improve file cache performance by causing hinted data to be kept in the cache when it would ordinarily be ejected by the cache's LRU replacement policy.) This section examines the prefetching performance of the SpecHint design by comparing the prefetching behavior of naive SpecHint executables with the prefetching behavior of both the original, non-hinting executables and the manual hinting executables.

Tables 8.3 and 8.4 show the number of prefetch I/Os, and the number of prefetched blocks which were fully prefetched before being requested during normal execution, only partially prefetched before being requested during normal execution, or prefetched uselessly (i.e. ejected from the cache without ever being used). The sum of the fully and partially prefetched blocks is the number of usefully prefetched blocks. For the original, non-hinting executables, prefetches occur only as a result of the system's automatic readahead policy. For the hinting executables, prefetches occur as a result of both the readahead policy and hint-guided prefetching. Therefore, even if a hinting executable issues no incorrect hints, it may accrue some incorrectly prefetched blocks. Since hinted reads do not

1 Disk							
Benchmark	Version	Prefetch	Blocks prefetched				
		I/Os	Fully +	Partially	= Usefully	Uselessly	
Agrep	Original	526	506	502	1009	3	(< 1%)
	Naive	1746	2539	453	2992	16	(< 1%)
	Manual	1664	2507	423	2931	2	(< 1%)
XDataSlice	Original	22721	12649	12744	25394	35260	(58%)
	Naive	16148	32174	12900	45075	68	(< 1%)
	Manual	14945	33054	11820	44874	15	(< 1%)
Gnuld	Original	2611	2467	2091	4559	1155	(20%)
	Naive	5318	2294	6772	9067	537	(6%)
	Manual	4128	8129	1977	10107	7	(< 1%)
Postgres 80%	Original	247	710	93	804	349	(30%)
	Naive	13481	8443	4217	12660	1485	(11%)
	Manual	8991	12285	824	13109	1651	(11%)
Postgres 20%	Original	208	901	65	967	96	(9%)
	Naive	4161	2553	1692	4246	710	(14%)
	Manual	3303	4269	286	4556	272	(6%)
Sphinx	Original	4515	14881	3204	18085	3448	(16%)
	Naive	7649	10282	10541	20823	1521	(7%)
	Manual	6626	18462	8271	26734	115	(< 1%)

Table 8.3: Prefetching performance of the original, naive SpecHint, and manually modified applications on one disk. All applications experience automatic prefetching in the form of readahead. The hinting applications also experience hint-driven prefetching. *Fully* is the number of prefetched blocks that were fully prefetched before being requested by normal execution. *Partially* is the number of prefetched blocks that were only partially prefetched before being requested by normal execution. *Usefully* is the sum of *Fully* and *Partially*. Finally, *Uselessly* is the number and percentage of prefetched blocks that were not used before being ejected from the cache.

trigger the readahead policy, a successfully hinting executable can avoid much unnecessary readahead. Results are shown for both the one and four disk configurations because prefetching performance is highly dependent on the available I/O bandwidth,

Whenever normal execution requests a blocks that has only been partially prefetched, it will experience less I/O latency than if no prefetch had been initiated, but will still need to stall until the fetch completes. Therefore, the difference in the number of blocks fully prefetched by the naive SpecHint and manual hinting executables show how often speculative execution was not able to generate hints early enough to hide the same amount of I/O latency as the manual hinting executables. As expected given the overall results, the number of blocks fully prefetched for Agrep and XDataSlice is essentially the same for the naive SpecHint and manual hinting executables. For the other benchmarks, even when the naive SpecHint executable correctly prefetches a similar number of blocks as the manual hinting executable, the number of blocks it is able to prefetch fully is substantially less than

4 Disks							
Benchmark	Version	Prefetch I/Os	Blocks prefetched Fully +	Partially	= Usefully	Uselessly	
Agrep	Original	533	542	482	1024	3	(< 1%)
	Naive	1759	2791	198	2989	16	(< 1%)
	Manual	1647	2754	176	2931	2	(< 1%)
XDataSlice	Original	22712	12753	12656	25410	35189	(58%)
	Naive	16159	40337	4776	45114	84	(< 1%)
	Manual	14941	40427	4421	44849	12	(< 1%)
Gnuld	Original	2656	2491	2054	4545	1181	(21%)
	Naive	5285	3739	5176	8915	442	(5%)
	Manual	4157	8922	1020	9942	7	(< 1%)
Postgres 80%	Original	238	700	85	786	346	(31%)
	Naive	12081	3614	8090	11704	1075	(8%)
	Manual	8747	12554	373	12927	1690	(12%)
Postgres 20%	Original	210	950	68	1018	78	(7%)
	Naive	3800	1742	2314	4056	592	(13%)
	Manual	3253	4424	192	4616	246	(5%)
Sphinx	Original	4422	15893	2073	17967	3287	(15%)
	Naive	5778	14247	6256	20503	1761	(8%)
	Manual	6440	22510	4062	26572	106	(< 1%)

Table 8.4: Prefetching performance of the original, naive SpecHint, and manually modified applications on four disks.

the number of blocks that the manual hinting executable is able to prefetch fully. Therefore, these figures reveal that one of the main reason for the performance difference between the naive SpecHint and manual hinting executables is that the naive SpecHint executables often fail to issue hints early enough to receive the same benefit from prefetching.

Looking at the prefetching performance of the original, non-hinting executables reveals the performance of the operating system's readahead policy. For Agrep, readahead is not effective because, although each data file is read sequentially, most of the files are too small to benefit from readahead. Readahead is not triggered until two sequential reads have occurred, but (as shown in Figure 7.2) 80% of Agrep's data files fit in one or two blocks. For XDataSlice, readahead actually hurts performance because the strided access pattern generally reads enough data sequentially to trigger bandwidth-consuming readahead, but then strides to a different location in the data file. Notice that, even if an operating system were modified such that it could detect strided accesses, it would have a hard time automating prefetching for XDataSlice's strided accesses since, as explained in Section 7.3, XDataSlice's striding pattern cannot be expressed with a single stride. For Gnuld and the Postgres benchmarks, readahead is neither particularly helpful nor harmful. Finally, readahead is very helpful for Sphinx since most of the read calls in the Sphinx benchmark are sequential reads of initialization files, as described in Section 7.3.

The hinting executables are able to improve the prefetching performance of Agrep by issuing hints across files. They improve the performance of XDataSlice both by issuing hints for the correct accesses, and by stopping unnecessary readahead. For Gnuld, Postgres and Sphinx, they issue hints for accesses that look random from the perspective of the operating system. The manually modified Sphinx issues hints for the sequential reads even though its performance would not change if it did not issue those hints. The speculating Sphinx actually fully prefetches *fewer* blocks than the original, non-hinting Sphinx because it issues hints for some, but not all, of these sequential reads. Since hinted reads do not trigger readahead, the data for sequential reads after these hinted reads are not prefetched by readahead. This explains why speculating Sphinx is as or less effective at prefetching than the original, non-hinting Sphinx.

Comparing the prefetching performance of the one and four disk configurations reveals the performance advantage of multiple disks. For both the naive SpecHint and manual hinting executables, the number of blocks that can be prefetched fully generally increases with the number of disks because the system is able to exploit more I/O parallelism to service prefetches at a higher rate. The speculating Postgres benchmarks are an exception. Increasing the rate at which prefetches are serviced can reduce the I/O stall time experienced by the original thread, causing the speculating thread to have less opportunity to speculate. For the Postgres benchmarks, this causes the speculating thread to issue many hints much later on four disks than on a single disk. Finally, as expected given the relative overall performance of the naive SpecHint and manual hinting executables for one and four disks, the prefetching performance of the naive SpecHint executables tends to be more sensitive to fewer disks. This is consistent with the fact that the naive SpecHint executables are not able to generate hints as early as the manual hinting executables, and are therefore more sensitive to limited I/O bandwidth.

Clustering multiple contiguous accesses reduces the processing overhead of issuing those accesses and improves disk throughput. As mentioned earlier, the operating system automatically clusters readahead requests, and TIP automatically clusters hinted prefetches. The more hints are outstanding at any time, the more likely there will be hints that TIP can cluster. The number of prefetch I/Os shows that the manual hinting executables are much more successful at issuing hints early enough to benefit from clustering; in many cases, the manual hinting executables correctly prefetch more blocks than the naive SpecHint executables while issuing fewer I/O requests. Notice that prefetches for the original, non-hinting Sphinx are clustered more successfully than for the speculating Sphinx since, as discussed in a prior paragraph, the hints issued by the speculating Sphinx disrupt clusterable readahead.

Finally, notice that the number of correctly and incorrectly hinted blocks is not necessarily indicative of the number of correctly and incorrectly prefetched blocks. There are two main reasons for this. First, hints will only trigger prefetches if the hinted data is not already cached. Therefore, for example, even though the speculating Postgres 80% issues many more incorrect hints in the one disk case, it does not incorrectly prefetch nearly as many additional blocks because most of these incorrect hints specify cached data. Second,

1 Disk						
Benchmark	Version	Block requests	Page faults	Block reuses	Unprefetched misses	Unprefetched miss I/Os
Agrep	Original	3059	135	108	1941	1932
	Naive	3503	578	449	61	37
	Manual	3072	148	118	22	11
XDataSlice	Original	48710	2349	3136	20179	20101
	Naive	49293	2935	3852	366	221
	Manual	48554	2196	3489	190	138
Gnuld	Original	22981	554	11523	6898	5110
	Naive	24151	1689	12799	2284	1809
	Manual	22986	559	12609	269	249
Postgres 80%	Original	33776	2513	20465	12507	12415
	Naive	36828	4252	22879	1288	1098
	Manual	33810	2540	19433	1267	1174
Postgres 20%	Original	10727	2030	5601	4158	4088
	Naive	12573	3620	7211	1115	951
	Manual	10765	2061	5193	1015	946
Sphinx	Original	79136	1402	51259	9791	4540
	Naive	80174	2394	51868	7482	4145
	Manual	78972	1471	51838	399	315

Table 8.5: Cache performance of the original, naive SpecHint, and manually modified applications on one disk. *Block requests* is the total number of blocks requests to the cache, not including requests for empty blocks. *Page faults* is the number of block requests for mapped blocks. *Block reuses* is the number of block requests for blocks in the cache that have already been used at least once, and *Unprefetched misses* is the number block requests for blocks not already in the cache and not in the process of being prefetched. *Unprefetched miss I/Os* is the number of I/Os issued to obtain these blocks. The total number of read I/Os can be calculated as *Unprefetched miss I/Os + Prefetch I/Os* (where the latter term can be found in Table 8.3).

non-hinted accesses still trigger the operating system's automatic readahead policy. Therefore, for example, even though the speculating Sphinx issues many more correct hints on a single disk, the number of correctly prefetched blocks is about the same regardless of the number of disks. This occurs because the additional hints are for blocks which, if not hinted, would still be prefetched by the automatic readahead policy.

8.1.4 Caching performance

The SpecHint design will cause an increase in memory pressure, which could lead to more page faults. Also, uselessly prefetched blocks could cause data to be ejected from the cache prematurely, so that additional I/Os are necessary to refetch the ejected data. Therefore, to understand the I/O stalls experienced by an execution, it is necessary to look at not only

4 Disks						
`Benchmark`	**Version**	**Block requests**	**Page faults**	**Block reuses**	**Unprefetched misses**	**Unprefetched miss I/Os**
`Agrep`	Original	3059	135	108	1926	1917
	Naive	3503	578	449	64	40
	Manual	3072	148	118	23	11
`XDataSlice`	Original	48671	2313	3097	20164	20097
	Naive	49250	2885	3755	380	228
	Manual	48521	2162	3473	197	136
`Gnuld`	Original	22998	571	11520	6932	5128
	Naive	24145	1684	12789	2441	1919
	Manual	22986	559	12744	299	284
`Postgres 80%`	Original	33751	2489	20291	12674	12566
	Naive	36077	4205	22191	2181	2014
	Manual	33844	2573	19636	1279	1174
`Postgres 20%`	Original	10876	2040	5680	4177	4098
	Naive	12599	3581	7159	1383	1253
	Manual	10927	2076	5307	1003	927
`Sphinx`	Original	79127	1393	51372	9787	4520
	Naive	80213	2459	52042	7667	3976
	Manual	78992	1491	52017	402	309

Table 8.6: Cache performance of the original, naive SpecHint, and manually modified applications on four disks.

on its prefetching performance, but also its caching performance. This section examines the caching performance of the SpecHint design by comparing the caching performance of naive SpecHint executables against that of both the original, non-hinting executables and the manual hinting executables.

Tables 8.5 and 8.6 show the total number of requests for non-empty cache blocks (*Block requests*), and the number of such requests for mapped blocks (*Page faults*). It also shows the number of such requests for blocks in the cache that have already been accessed (*Block reuses*), the number of such requests for blocks both not in the cache and not in the process of being prefetched (*Unprefetched misses*), and the number of I/Os issued to obtain these unprefetched missing blocks (*Unprefetched miss I/Os*). The last two differ because requests for multiple blocks are sometimes coalesced into a single I/O. The I/O stall experienced by an execution will be the remaining I/O latency for accesses to partially prefetched blocks, plus the full I/O latency for the *Unprefetched misses*. Figures are given for the one and four disk cases, corresponding to the prefetching results in Tables 8.3 and 8.4.

As expected given the addition of shadow code and data structures to support speculative execution, the number of page faults and the total number of block requests to the file cache is higher for the naive SpecHint executables. This increase is matched or at least offset, however, by an increase in the number of block reuses. The increase in the

4 Disks						
`Benchmark`	**Version**	**User time (s)**	**System time (s)**	**Soft page faults**	**Hard page faults**	**Memory footprint (KB)**
`Agrep`	Original	0.4	1.4	39	4	160
	Naive	0.4	1.2	100	20	928
`XDataSlice`	Original	8.1	26.4	8106	58	63496
	Naive	8.0	18.7	8110	60	65648
`Gnuld`	Original	3.5	5.3	1343	11	10376
	Naive	4.3	4.2	1680	30	15464
`Postgres 80%`	Original	31.7	12.6	466	78	2376
	Naive	31.4	10.2	500	82	5320
`Postgres 20%`	Original	19.7	3.9	435	56	2472
	Naive	20.6	3.3	474	73	5408
`Sphinx`	Original	136.7	12.7	11203	81	82960
	Naive	140.2	12.8	11063	87	86472

Table 8.7: Processor and memory performance of the original, non-hinting applications and normal execution in the naive SpecHint applications on four disks. Except for *Memory footprint*, all figures for naive SpecHint applications capture the performance of only the original thread. *Memory footprint* is the maximum amount of memory resident for the application at any time during its execution. *Soft page faults* require operating system intervention, but no I/O, while *hard page faults* require I/O.

number of block reuses also indicates that, while useless prefetches by the naive SpecHint executables may sometimes hurt cache performance, this effect is not large. Notice that, in all cases, the naive SpecHint executables experience fewer unprefetched misses than the original executables (and more than the manual hinting executables).

8.1.5 Overhead to normal execution

One of the three design goals was to avoid hurting application performance. This goal affected many elements of the SpecHint design. For example, the minimal scheduling priority of the speculating thread and the reliance on TIP's cost-benefit management of cache resources attempt to prevent speculative execution from hurting application performance. Other elements of the design, like the creation of shadow code and the resynchronization policy and mechanism, attempt to avoid hurting the performance of normal execution. The previous section demonstrated that the cache replacement performance of the naive SpecHint executables is similar to the cache replacement performance of the original executables. This section examines the amount of work that the SpecHint design adds to normal execution by comparing the processor and memory performance of the original thread in naive SpecHint executables to that of the original, non-hinting executables.

Table 8.7 shows processor times, number of page faults, and the maximum memory footprint. Except for the maximum memory footprint figures, the figures for the naive

SpecHint executables reflect the performance of only the original thread, so they can be used to determine the overhead incurred by the target normal execution (which is the only normal execution in these single application tests). Figures are given for the four disk case only; since these factors are independent of the amount of available I/O bandwidth, the results are similar for other numbers of disks.

The SpecHint design increases the user time of normal execution by requiring the original thread to execute additional code to support speculative execution (as discussed in Section 6.3.2). Looking at the user times figures, we see that, as intended, the user time of normal execution increases by at most a negligible amount when compared with the elapsed times of executing these benchmarks. The SpecHint design increases the system time of normal execution by the processing cost of servicing additional soft and hard page faults. However, it can also decrease the system time of normal execution by allowing the speculating thread to pay the processing cost of issuing I/Os by issuing correct prefetches. Looking at the system time figures, we see that this decrease outweighs any increase due to additional page faults in all the benchmarks except Sphinx (for which there is a insignificant increase). Sphinx is an exception because, as already discussed, the speculating Sphinx is not very successful at generating correct prefetches.

As indicated by the larger memory footprints, the naive SpecHint executables increase memory pressure substantially. This is not surprising given that the speculating thread accesses a different set of code pages and several fairly large speculative execution support data structures, performs copy-on-write, and dynamically allocates memory while speculating. This table shows how the increase in memory pressure leads to a substantial increase in the number of page faults experienced by the original thread. Most of the increase, however, is in soft page faults, which require operating system intervention but no additional I/O. The increase in the number of hard page faults (which require I/O) is much smaller, but still substantial in several cases. Notice that it is conceivable that SpecHint could instead decrease the number of page faults experienced during normal execution. In particular, as the speculating thread executes, it could fault in the data structures that will be accessed during subsequent normal execution. With these benchmarks and the current implementation, however, such an effect is not apparent.

To determine the source of the additional page faults, I conducted tests in which the speculating thread exits immediately. Comparing the number of page faults generated during these tests against the numbers shown above revealed that, while the increase in soft page faults is mainly the result of speculative execution increasing memory pressure, the increase in hard page faults is independent of actually performing speculative execution. Instead, it is an artifact of my implementation. Specifically, for ease of implementation, my research implementation intersperses additional data with the original data, so the original data is spread over more pages. It is actually a safety violation for the implementation to relocate the original data because the original code needs to be updated to reflect changes in data addresses. A non-research implement should instead place all the additional data in a separate location. This should not present any technical difficulties and would not only satisfy the safety goals, but should also decrease or even eliminate additional hard page

4 Disks							
Benchmark	**Median measured dilation**	**Copies per resynch**	**Resynchronizations Number**	**Resynchronizations Ave time (us)**	**Soft faults**	**Hard faults**	**Signals**
Agrep	9.1	1	4	219	59	2	0
XDataSlice	2.8	26	68	142	346	34	4
Gnuld	2.0	14	1697	68	627	23	253
Postgres 80%	8.0	50	2082	142	450	44	0
Postgres 20%	5.8	46	1114	157	434	37	1
Sphinx	5.1	25	3938	186	833	31	139

Table 8.8: Performance characteristics of speculative execution on four disks. The dilation factor is the ratio between the speed of speculative execution and normal execution. The process used to calculate the *median measured dilation* factor is described in the text. *Copies per resynch* is the average number of software copy-on-write region copies performed per resynchronization, with a region size of 1KB. *Resynchronizations* gives the number of resynchronizations performed by the speculating thread, and the average time to perform a single resynchronization. *Soft faults*, *hard faults* and *signals* are the number of soft page faults, hard page faults, and signals generated by the speculating thread. The original, non-hinting applications (and, therefore, the original thread in the speculating applications) generate no signals.

faults experienced during normal execution.

8.1.6 Performance of speculative execution

This section examines the behavior of the speculating thread in order to provide more insight into what occurs during speculative execution. This includes an evaluation of the cost of the safety mechanisms in the SpecHint design, as well as the behavior of the resynchronization policy and mechanism.

The *dilation factor* is the ratio between the speed of speculative execution and normal execution. As such, it is useful because it captures the processing cost of work added to speculative execution. This work mostly consists of software copy-on-write checks, but also includes any actual software copy-on-write copying, stack pointer checks, target address checks, and executions of the support routines for the speculating thread. To measure the dilation factor, a statistics-gathering version of each naive SpecHint executable logged the time that the speculating thread spent between each pair of consecutive hint calls between which no reynchronization occurred, and which were both correct. It also logged the time that the original thread spent between each corresponding pair of consecutive read calls. The median measured dilation factor is then calculated as the median of these ratios for each benchmark. Medians are given rather than averages because a few very large inter-hint times and inter-hint-time to inter-read-time ratios were recorded. These abnormally large values skew the averages. Notice that the calculation of the median measured dilation factor does not include the dilation factor during inter-read code that prevents the current speculation from generating correct hints for future read requests (either because the code is expensive, or because it contains dependencies on data unavailable during speculative execution). Therefore, the median measured dilation factor is known to be accurate only for Agrep and XDataSlice. For the other benchmarks, the true dilation factor is probably higher by a benchmark-dependent amount.

Table 8.8 shows the median measured dilation factor for each benchmark. The figures in the table indicate that, as expected, the processor cost of performing software copy-on-write checks is substantial in all cases. It should be possible to reduce the dilation factors in a more optimized implementation of the SpecHint design by performing static optimizations during the binary modification process to reduce the number of software copy-on-write checks that will be executed, or reduce the cost of the check that must be performed. For example, loop blocking (a well-known loop transformation) could be applied to loops that contain a memory instruction whose address is incremented by a constant factor each iteration such that a copy-on-write check would only be performed for that instruction on the first iteration, and whenever the incrementing causes the address to cross a region boundary.

To reveal the cost of software copy-on-write copying and resynchronization, the table also shows the average number of software copy-on-write memory regions that the speculating thread copies per resynchronization, the number of times it resynchronizes, and the average amount of time it takes to perform a resynchronization. Since it takes less than 3 us to copy a 1KB region on my testbed, the average number of copies per resynchronization indicate that the processing cost of actually copying memory regions is very small. The figures also show that the average time for a single resynchronization is very small for all the benchmarks. This demonstrates one of the claims that underly the SpecHint design: that using threads to implement speculative execution would allow fast resynchronization. The resynchronization time varies by a factor of three across the benchmarks because it depends on the size of the original thread's stack at the time of resynchronization (since the speculating thread makes a copy of the original thread's stack) and the dynamic memory allocations of the speculating thread (since the speculating thread cleans up its dynamic memory allocation state during each resynchronization).

Relative to the number of read calls (which is an upper bound on the number of resynchronizations), the number of resynchronizations is much lower for Agrep and XDataSlice than for any of the other benchmarks. This makes sense because speculative execution is so successful at generating correct hints for Agrep and XDataSlice, and resynchronization only occurs when speculative execution falls behind normal execution, or strays from the correct execution path. Notice that no conclusions can be drawn by comparing the figures in the table for measured dilation factors and average number of regions copied per resynchronization – in fact, the results of such a comparison seem counter-intuitive – because, as explained in a prior paragraph, the dilation factor figures are not representative of all of the speculative execution that occurs.

The copy-on-write region size used to obtain all the results shown in this dissertation was 1KB. One potential advantage of the software copy-on-write mechanism is that it allows a copy-on-write region size different from the operating system's page size. To examine whether this added flexibility is truly an advantage, I conducted experiments with other region sizes. Logically, a larger region size will probably result in more unnecessary copying of data that is not updated, but in the same region as some data that is updated. A smaller region size, on the other hand, will increase the number of region identifiers. For some implementations, this could greatly increase the amount of memory needed to implement software copy-on-write. In my implementation, it could hurt the caching performance of a small (3-entry) copy-on-write checking cache I use to speed consecutive checks for addresses in the same regions. Results for the experiments I conducted with other region sizes are not shown, but the basic findings were that performance is not particularly sensitive to the copy-on-write region size for powers of 2 between 512B and 4KB, and only slightly degrades with a region size of 128B or 8KB. Since most systems use a page size of either 4KB or 8 KB, this indicates that relying on standard operating system/hardware copy-on-write in, for example, an in-kernel implementation (as sketched in Section 3.2.2) would not be a performance problem.

Finally, the table shows the number of page faults and signals generated by the speculating thread. The speculating thread experiences a substantial number of both soft and hard page faults to access its code and data. The speculating thread also generates signals in many of the benchmarks. This is not surprising since the speculating thread sometimes uses incorrect data values. The speculating thread generates SIGFPE signals during XDataSlice by attempting to divide by zero, and generates SIGSEGV signals during Gnuld, Postgres and Sphinx by attempting to load from inaccessible memory addresses.

8.2 Single application, improving SpecHint effectiveness

As previously discussed, there are two fundamental reasons that generating prefetches automatically through speculative execution may be less effective than manually modifying applications to generate prefetches. First, speculative execution can be misled by stale data values (e.g. in read buffers of incomplete read calls) such that it may not be able to generate as many correct hints and may generate incorrect hints that hurt performance. Second, speculative execution requires spare processing and memory resources. When there are insufficient spare processing cycles, it may not be able to generate as many correct hints, or may not generate correct hints early enough to hide the same amount of I/O latency. When there is inadequate memory, it may inadvertently cause useful data to be prematurely ejected from memory, so that additional I/Os are required to refetch such ejected data.

This section examines the impact, in single application tests, of various mechanisms that attempt to improve the effectiveness of naive SpecHint. Specifically, the section evaluates mechanisms for decreasing the amount of memory used during speculative execution, for filtering potential hints based on predictions of how likely they are to be correct, for attempting to increase the effectivness of stale data values, and for attempting to reduce the amount of work that speculating threads must perform to generate correct hints. For each mechanism, it examines the tradeoff between the amount of resources consumed by the mechanism, and the benefit provided by the mechanism. Notice that, in single application tests, it is impossible for these mechanisms to improve the performance of Agrep or XDataSlice since naive SpecHint already provides near-optimal performance for these benchmarks. Results for these benchmarks are included in this section only to help evaluate the cost of supporting the mechanisms.

8.2.1 Minimal updates to execution state

As shown in Table 8.8, software copy-on-write adds a substantial amount of work to speculative execution. One alternative would be to constrain the updates that speculative execution is allowed to make to its execution state such that software copy-on-write is no longer necessary. In particular, I modified the SpecHint tool such that it, while copying the original code to begin creating shadow code, the tool removed all store instructions not off the stack pointer. Store instructions off the stack pointer were allowed in shadow code since, as discussed in Section 6.3.1.1, stack pointer checks ensure that all such stores executed would store to the memory allocated to hold the speculating thread's stack. In addition, the tool no longer adds load checks before load instructions in shadow code; they are no longer necessary since no store checks will be executed, so non-shadow data regions will never be copied (so there will never be copies to which loads should be redirected).

Table 8.9 shows elapsed times relative to original non-hinting executables for binaries with which speculating threads are allowed to update only register and stack values (*MinUpdates*), and the naive SpecHint binaries. The results show that constraining updates in this fashion is sufficient for Agrep and Gnuld; in fact, slightly larger performance benefits were obtained for thse benchmarks because the elimination of copy-on-write and store instructions increased the speed with which speculative execution could generate hints. On the other hand, constraining updates eliminated the benefit for Postgres, and incurred a substantial penalty for XDataSlice. Both of these applications determine their future data requests by manipulating (i.e. not just loading) data not stored in the stack. The performance of XDataSlice degraded substantially because large numbers of incorrect hints were generated, causing large numbers of incorrect prefetches to be issued, and these prefetches

4 Disks						
	Elapsed time relative to original non-hinting applications					
	Agrep	XDataSlice	Gnuld	Postgres 80%	Postgres 20%	Sphinx
MinUpdates	31%	135%	61%	100%	102%	102%
Naive	32%	29%	64%	64%	75%	100%

Table 8.9: Elapsed time relative to original non-hinting applications. *MinUpdates* shows results for when speculative execution is allowed to update only register and stack values. *Naive* results are repeated here to facilitate comparison. Results shown are for the four disk configuration.

delayed demand I/Os. These results indicate that, while constraining updates to register and stack values may be effective for some applications, it reduces the scope of the tool.

8.2.2 Prefetch correctness prediction and filtering

As shown in Table 8.2, the manual hinting executables rarely generate incorrect hints because a good programmer can usually modify applications such that only correct hints will be generated. On the other hand, since speculative execution will sometimes use incorrect data values, speculating executables can generate large numbers of incorrect hints. This is a problem because incorrect hints can lead to useless prefetches, which can hurt performance in two ways: 1) by ejecting useful data from memory to make space for the unnecessarily prefetched data, so additional I/Os are necessary to obtain the data required by normal execution, and 2) by increasing contention for the disk heads such that I/Os for normal execution are delayed by servicing of useless prefetches.

The hint correctness prediction and filtering mechanism attempts to eliminate the performance loss of speculative execution due to incorrect hints. As described in Sections 5.1.4 and 5.2.2, it consists of a history-based technique for predicting the probability that a potential prefetch will be correct, and a threshhold-based technique for deciding whether to issue a potential hint with a certain predicted correctness probability. The key assumption is that the code path executed in order to discover a potential hint will often be correlated to whether or not the hint will prove to be correct. The implementation evaluated in this chapter identifies the code path executed in order to discover a potential hint as a simple function of the current stack pointer and the top 3 return addresses on the filtering return address stack upon discovering the potential hint.

Table 8.10 shows how adding this mechanism changes the hints issued by the speculating executables when their data resides on a single disk. The single disk figures are most relevant because incorrect prefetches are most likely to delay demand requests when there is only a single disk (and therefore no possibility of servicing multiple I/O requests in parallel). The table shows that the mechanism succeeds at greatly reducing the number of incorrect hint calls (and incorrectly hinted blocks). However, the table also shows that the mechanism slightly but noticeably decreases the number of correctly hinted calls and blocks for Postgres 20% and Sphinx. This is partially due to the cost of maintaining the filtering return address stack, which slows speculative execution so that it is sometimes unable to issue as many hints, but is mainly due to accidentally filtering out an occasional hint that would have been correct.

While the reduction of incorrect hints looks somewhat promising, the end goal is to decrease application elapsed times. Figure 8.2 shows how adding the prefetch correctness prediction and filtering mechanism to naive SpecHint changes elapsed times. In particular, the Y axes of these graphs give, for each of executable version, *V*: (Time$_{Naive}$ - Time$_{V}$) / Time$_{Original}$, expressed as a percentage. *SpecHint, filter overhead* executables perform all the work necessary to add this mechanism, but still issue all potential hints. *SpecHint, filter* executables actually filter potential hints. Therefore, *SpecHint, filter overhead* indicates

1 Disk							
Benchmark	Version	Correctly hinted				Incorrectly hinted	
		Calls		Blocks		Calls	Blocks
Agrep	Naive	2915	(68%)	2915	(100%)	0	0
	Filter	2914	(68%)	2914	(100%)	0	0
XDataSlice	Naive	45241	(98%)	45241	(98%)	0	0
	Filter	45239	(98%)	45239	(98%)	0	0
Gnuld	Naive	8215	(63%)	14659	(73%)	2266	2605
	Filter	8213	(63%)	14636	(73%)	272	395
Postgres 80%	Naive	14531	(47%)	14531	(47%)	8813	8813
	Filter	14677	(47%)	14677	(47%)	4045	4045
Postgres 20%	Naive	3922	(45%)	3922	(45%)	2903	2903
	Filter	3858	(44%)	3858	(44%)	729	729
Sphinx	Naive	10288	(16%)	16006	(21%)	164	256
	Filter	10018	(15%)	15648	(20%)	34	74

Table 8.10: Comparison of the hinting performance of SpecHint with the filtering mechanism and naive SpecHint on one disk. The table shows how many of the read calls (and blocks) were correctly hinted, as well as the number of incorrect hint calls (and blocks).

the cost of adding this mechanism; the difference between *SpecHint, filter* and *SpecHint, filter overhead* indicates the benefit of adding the mechanism; and *SpecHint, filter* by itself indicates the net benefit of adding the mechanism.

The results show that supporting the mechanism incurs at most a small cost; the largest increase in elapsed time relative to the original executable is less than 5%. The cost is greatest for the Postgres benchmarks because Postgres calls small procedures frequently, causing the cost of maintaining the filtering return address stack to be more noticeable.

The results also show that using the mechanism to actually filter hints out never hurts performance (i.e. *SpecHint, filter* is never slower than *SpecHint, filter overhead*). This indicates that any inadvertent filtering out of correct hints is always balanced or outweighed by the benefit of filtering out incorrect hints. On the other hand, even the small cost of supporting the mechanism does not appear worthwhile since the mechanism never produces a substantial benefit.

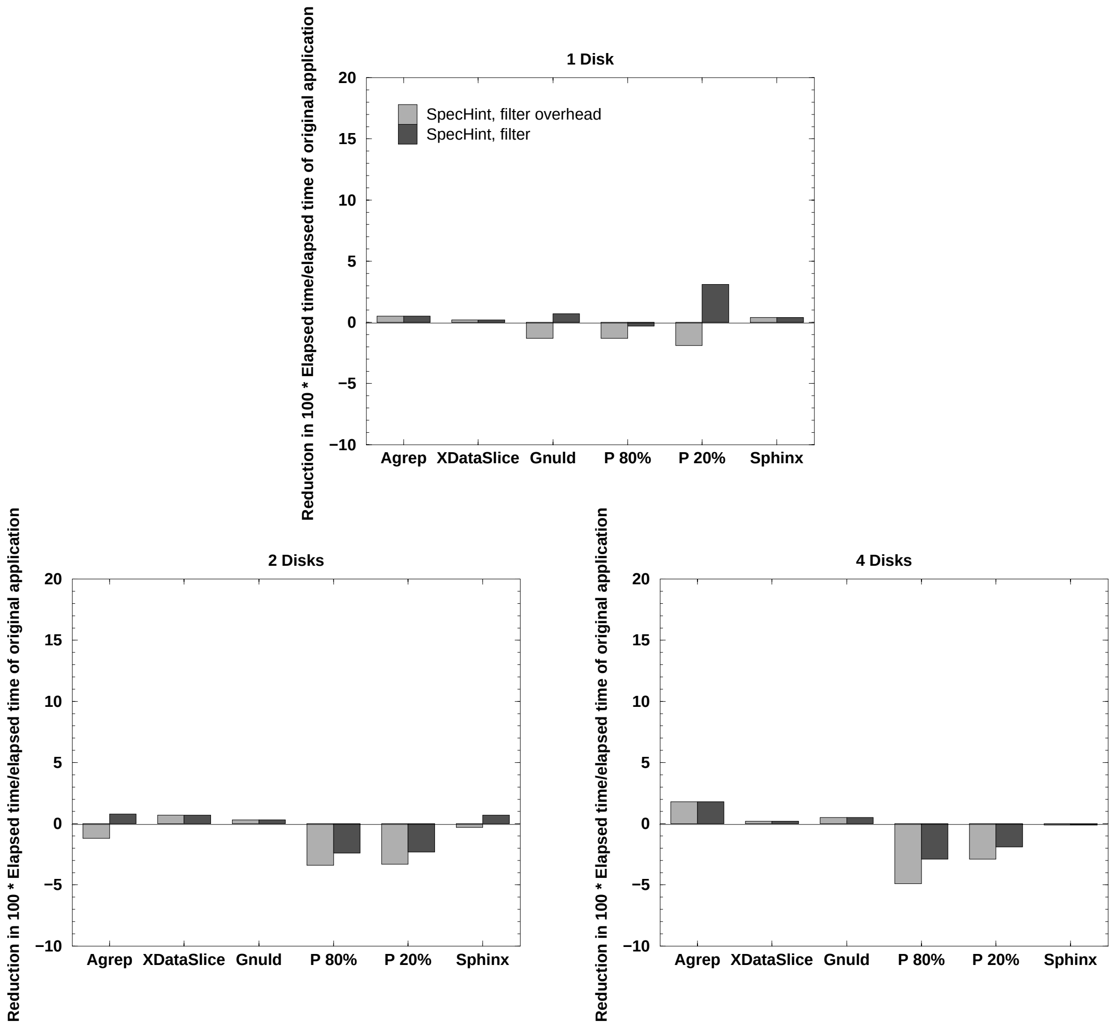

Figure 8.2: Performance impact of adding the hint correctness prediction and filtering mechanism. *SpecHint, filter overhead* results are for executables that perform all the work of the mechanism but still issue every potential hint. *SpecHint, filter* results are for executables that actually filter potential hints. The graphs show the reduction in the elapsed time of these executables relative to the elapsed time of naive SpecHint executables, expressed as a percentage of the elapsed time of the original, non-hinting executables. Results are shown for the one, two and four disk configurations.

8.2.3 Simple value prediction

Use of incorrect data values can not only cause speculative execution to generate incorrect hints, but also prevent speculative execution from generating correct hints. Incorrect data values originate from read buffers for read calls that have not yet completed. (They could also originate from system calls which are skipped during speculative execution, but this is not an issue in any of my benchmarks.) In the SpecHint design, when the speculating thread accesses a memory location in such a read buffer, it will simply acquire whatever value happens to reside in that memory location. It may then propagate this probably incorrect value during subsequent execution. As discussed in Sections 4.1.3 and 4.2.5, it may be possible to improve speculative execution by selecting values for these buffers that, while still possibly incorrect, are more likely to lead to faster or more accurate prefetching. There is a tradeoff, however, between how much extra processing and memory resources are needed to use more carefully selecting values during speculative execution, and how much benefit will be obtained as a result of using those more carefully selected values.

The simplest value prediction mechanism would be to pre-select a single value to predict. Such a mechanism would not add any overhead for selecting what values to predict, only overhead for causing the speculative execution to obtain the selected value when it attempts to load from a read buffer for an incomplete read call. Figure 8.3 shows how adding a simple single-value prediction mechanism to naive SpecHint changes the elapsed times of the speculating executables. *SpecHint, value overhead* indicates results when all the work of a single-value prediction mechanism is performed, but loads still obtain whatever essentially random value happens to be in read buffers of incomplete read calls. *SpecHint, clear* indicates results when all bits in read buffers of incomplete read calls are cleared (so that loading from such a buffer will yield an integer value of zero), *Set* shows elapsed time when all bits in such read buffers are set (i.e. an integer value of -1), and *One* indicates results when such read buffers are filled with the integer value of 1.

A naive implementation of a single-value prediction mechanism is for the speculating thread to allocate space for, and fill, a shadow data copy of the read buffer for the read call on which normal execution is blocked, and the read buffer for each subsequent read call (or, rather, hint call) encountered during speculative execution. This turns out to be very expensive, however, since the total amount of data read by each benchmark application is quite large. To reduce the cost of this mechanism, my measured implementation does not actually allocate and fill shadow data copies of read buffers. Instead, as discussed in Section 4.2.5, it leverages the software copy-on-write checks to provide approximately the same effect at a much lower cost. The *SpecHint, value overhead* results show that, even so, the added work to implement this mechanism incurs a small but noticeable cost, increasing the elapsed time relative to the original executable by up to 7%.

The differences between the results for each version of single-value prediction and the *SpecHint, value overhead* results indicate the benefit of that single-value prediction. These differences demonstrate that single-value prediction, for single values of integer 0, -1 or 1, fails to substantially change the elapsed time of any of the benchmarks. Therefore, the net "benefit" for these single values is negative in these tests. On the other hand, later

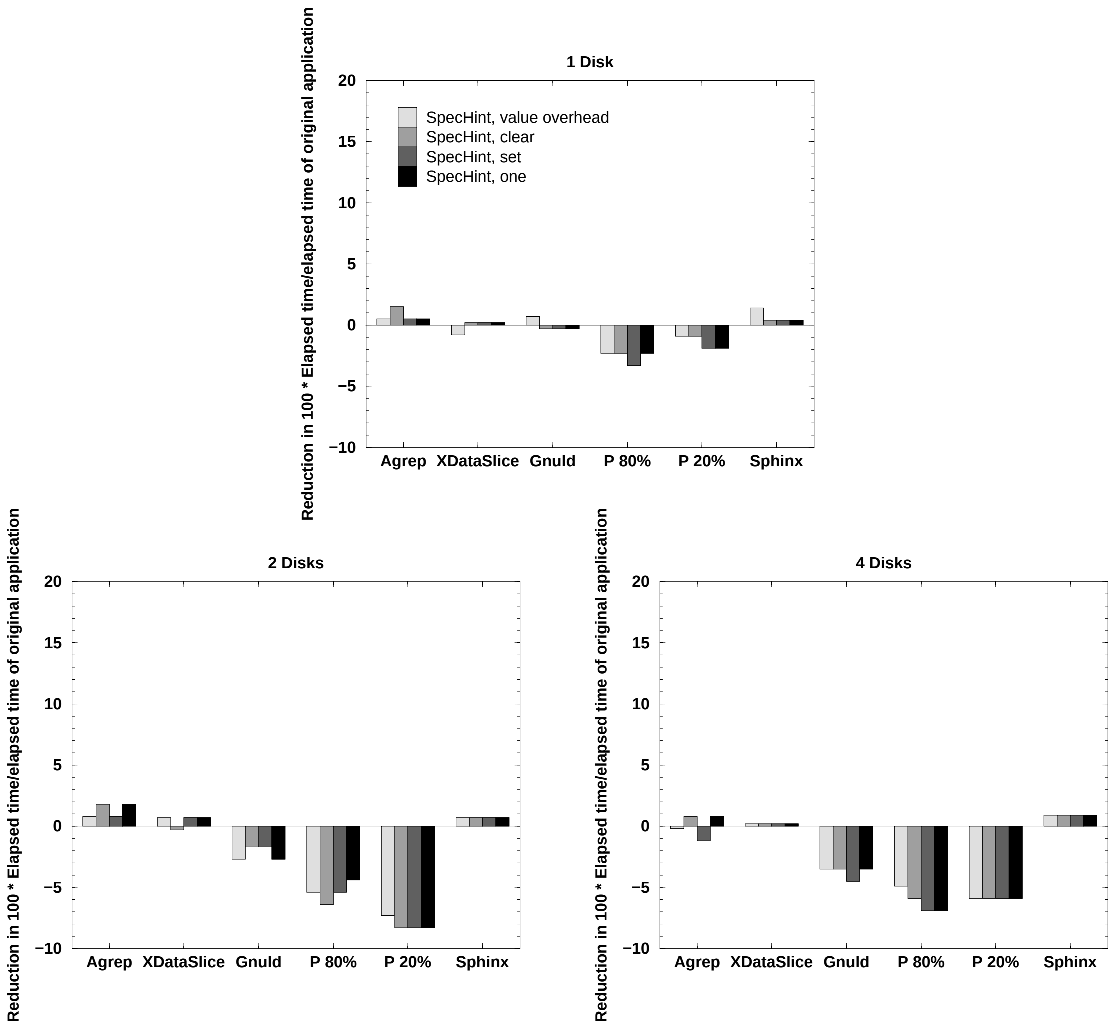

Figure 8.3: Performance impact of adding a simple value prediction mechanism. *SpecHint, value overhead* results are for executables that perform all the work of adding a single-value prediction mechanism, but still use the essentially random values in stale memory locations. *SpecHint, clear* and *SpecHint, set* results are for executables that logically clear and set the bits in stale memory locations, respectively (i.e. loading from such memory will return a zero or negative one, respectively, if interpretted as an integer). *SpecHint, one* results are for executables that logically fill stale memory locations with the integer value of one. The graphs show the reduction in the elapsed time of these executables relative to the elapsed time of naive SpecHint executables, expressed as a percentage of the elapsed time of the original, non-hinting executables. Results are shown for the one, two and four disk configurations.

sections in this chapter will reveal that, when combined with another optional mechanism, single-value prediction can sometimes provide substantial benefit.

8.2.4 Experimental slicing

The previous two mechanisms attempt to reduce the impact of incorrect values on speculative execution. The other reason that the prefetching performance of speculating executables is often not as good as the prefetching performance of manual hinting executables, is that speculative execution may require many processing cycles to generate hints (and is mainly restricted to consuming spare processing cycles). If there are insufficient spare processing cycles in which to perform speculative execution, a speculating executable will not be able to generate as many hints. Furthermore, even if there are sufficient spare processing cycles to generate the same number of hints, a speculating executable will generate hints later than the manual hinting executable, so its prefetches may not be able to hide as much I/O latency.

There will often, however, be a substantial amount of work performed during the execution of an application that is orthogonal to determining the stream of read requests issued by the application. Speculative execution does not need to perform this work to generate correct hints, and should avoid performing this work in order to reduce the amount of processing cycles and memory it needs to generate hints. Naive SpecHint makes no attempt to identify such unnecessary work. As discussed in Section 4.1.2, experimental slicing is a mechanism that enables speculative execution to identify and skip unnecessary work by dynamically testing how skipping some work changes the speed at which speculative execution can generate correct hints. As discussed in Section 4.2.4, my implementation of this mechanism enables the speculating thread to skip loops that dynamic monitoring and testing indicate would otherwise require many processing cycles, and decrease the speed at which the speculating thread would generate correct hints.

Figure 8.4 shows how adding the experimental slicing mechanism to naive SpecHint changes elapsed times. *SpecHint, slice overhead* indicates results when all the work necessary to implement the mechanism is performed, but no code is skipped during speculative execution. *SpecHint, slice* indicates results when the speculating thread will actually skip code.

The *SpecHint, slice overhead* results show that the work necessary to provide this mechanism never adds a noticeable amount of overhead. The *SpecHint, slice* results show that the mechanism provides a substantial benefit for Gnuld regardless of the number of disks, as well as a small but noticeable benefit for Sphinx on four disks. Closer examination of Gnuld revealed that the improved performance was the result of skipping a very small number of data-processing loops that are unnecessary for the generation of correct hints, but so expensive that naive SpecHint is never able to complete the loops and generate hints for subsequent reads.

8.2.5 Combining techniques

The previous three sections demonstrated that, of the three proposed mechanisms, only the experimental slicing mechanism can provide substantial performance benefits on its own. However, since these mechanisms are targetted at different problems, there is a chance that

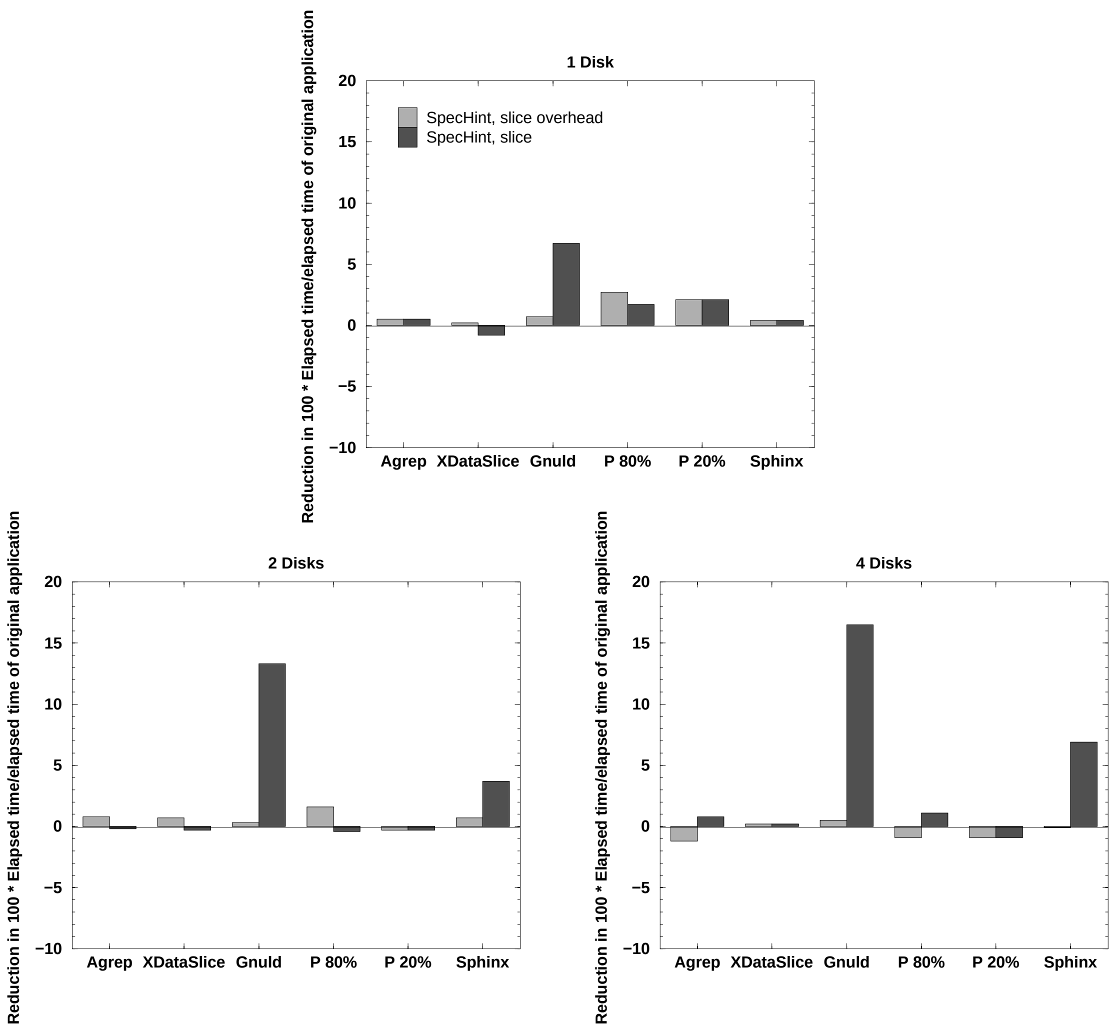

Figure 8.4: Performance impact of adding the experimental slicing mechanism. *SpecHint, slice overhead* results are for executables that perform all the work of the mechanism but do not skip any code during speculative execution. *SpecHint, slice* actually skips code during speculative execution. The graphs show the reduction in the elapsed time of these executables relative to the elapsed time of naive SpecHint executables, expressed as a percentage of the elapsed time of the original, non-hinting executables. Results are shown for the one, two and four disk configurations.

they will be complimentary. This section explores that possibility by examining performance when the mechanisms are added in different combinations.

Figure 8.5 shows how adding different combinations of these mechanisms changes the elapsed time of the benchmarks. Of the three versions of the single-value prediction mechanism, only *Clear* is used in these combinations because *Clear* had (marginally) the best performance in the single mechanism tests. The results demonstrate that the net benefit of these mechanisms is generally not additive. For example, combining *Filter* and *Slice* for Postgres 20% does not produce the combined benefit of *Filter* and *Slice*, only the same ben-

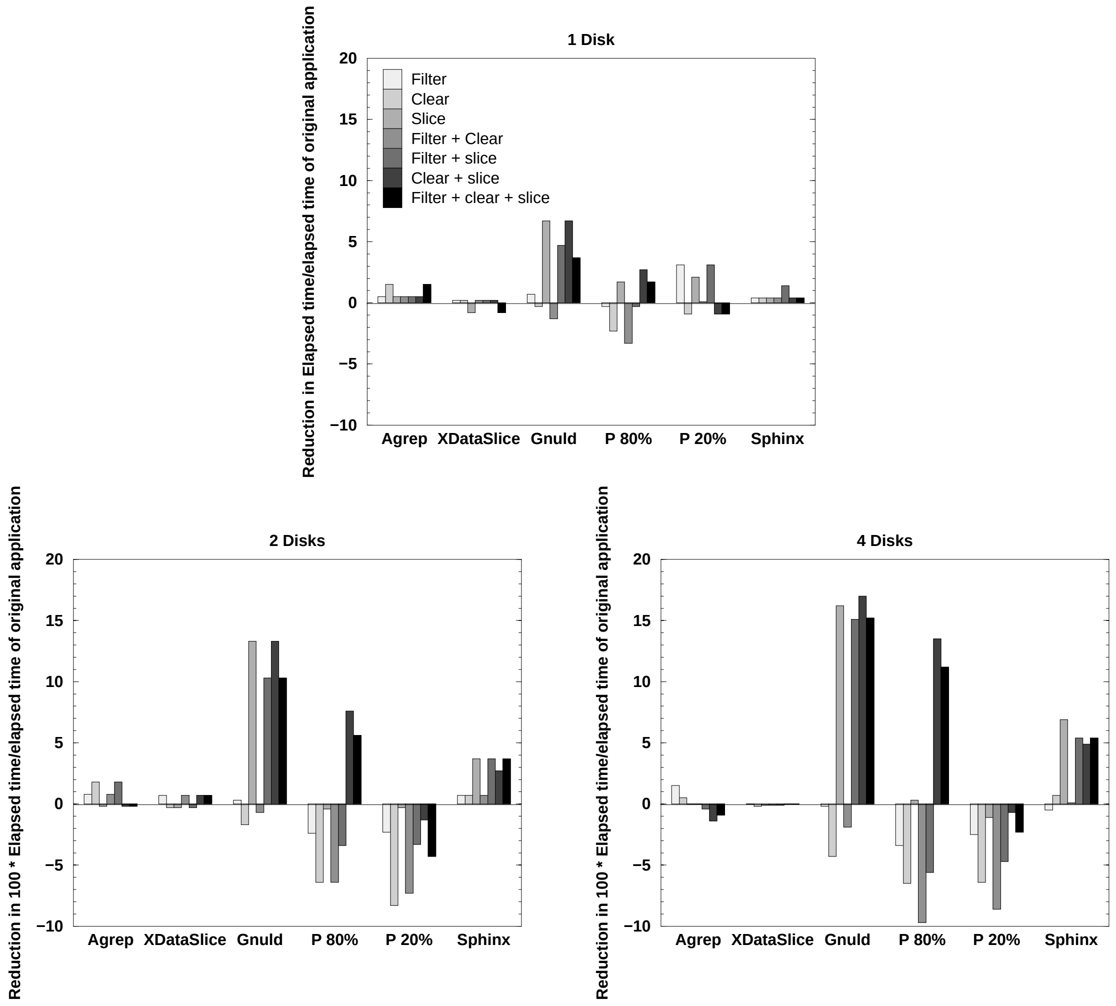

Figure 8.5: Performance impact of adding various combinations of the hint filtering, single-value prediction and experimental slicing mechanisms. The graphs show the reduction in the elapsed time of executables that use various combinations of these mechanisms relative to the elapsed time of naive SpecHint executables, expressed as a percentage of the elapsed time of the original, non-hinting executables.

efit as *Filter* by itself. On the other hand, the mechanisms are sometimes complimentary. In particular, combining *Clear* and *Slice* produces a substantial benefit for Postgres 80% on multi-disk systems, even though *Clear* by itself produces a substantial loss, and *Slice* by itself has a negligible effect. Closer examination of the benchmark reveals that *Slice* enables the speculating thread to skip a substantial amount of work, but skipping that work introduces additional stale data values that mislead the speculating thread. Adding *Clear* as well is sometimes sufficient to change these additional stale data values such that they no longer mislead the speculating thread. Therefore, when spare processing cycles are limited, the combination of these mechanisms can enable the speculating thread to generate correct

Disks	**Naive**	**Filter**	**Clear**	**Slice**	**Filter + Clear**	**Filter + Slice**	**Clear + Slice**	**Filter + Clear + Slice**	**Manual Hints**
1	0.90	0.90	0.91	0.89	0.91	0.89	0.89	0.89	0.80
2	0.67	0.67	0.69	0.65	0.69	0.65	0.63	0.65	0.53
4	0.55	0.56	0.57	0.52	0.58	0.54	0.50	0.51	0.39

Table 8.11: Geometric mean across benchmarks of the elapsed time relative to the original, non-hinting applications.

hints more quickly.

Table 8.11 summarizes the performance of the different options by presenting geometric means across the benchmarks of the elapsed times relative to the elapsed times of the original, non-hinting executables. The table shows that the combination of mechanisms which provides the best overall performance is *Clear* and *Slice*. Not only does this combination provide the largest average benefit, but also (unlike many of the other combinations) it never degrades the performance of naive SpecHint by a noticeable amount (the worst case is a less than 2% loss in the elapsed time relative to the original, non-hinting executable).

8.3 Projecting future performance

In this section, I project how the performance benefit of speculative execution will change assuming that the gap between processing speeds and disk access speeds continues to widen, and that file cache miss rates continue to stay about the same. As noted in Chapter 2, processing speeds have been increasing at around 58% a year, while disk access rates have only been increasing at around 8% a year, and file trace studies have indicated that file cache miss rates may not be decreasing despite increasing memory sizes.

Intuitively, as the gap widens, the number of cycles per I/O stall will increase, so the number of instructions into the future that the speculating thread will be able to reach during a given stall will increase. Therefore, the widening gap may enable the speculating thread to issue hints earlier. On the other hand, if all data requests after the *i*th future data request depend on data that is not available during the current speculation, then there will be no benefit to having more spare cycles with which to continue the current speculation after generating a hint for that *i*th future data request. Therefore, speculative execution may not always be able to capitalize on the additional processing opportunity.

To determine which of these effects will predominate for my benchmarks, I simulated the effect of a wider processor to disk access speed gap by having the striper delay I/O requests. First, the striper limits the number of outstanding requests per disk to one. Then, to simulate multiplying the gap on my testbed by a factor of n, if the last request to a disk was issued to the disk at time t_b and completed at time t_e, then the striper delays issuing a new request to the disk until time $t_b + n * (t_e - t_b)$. The striper also delays notifying

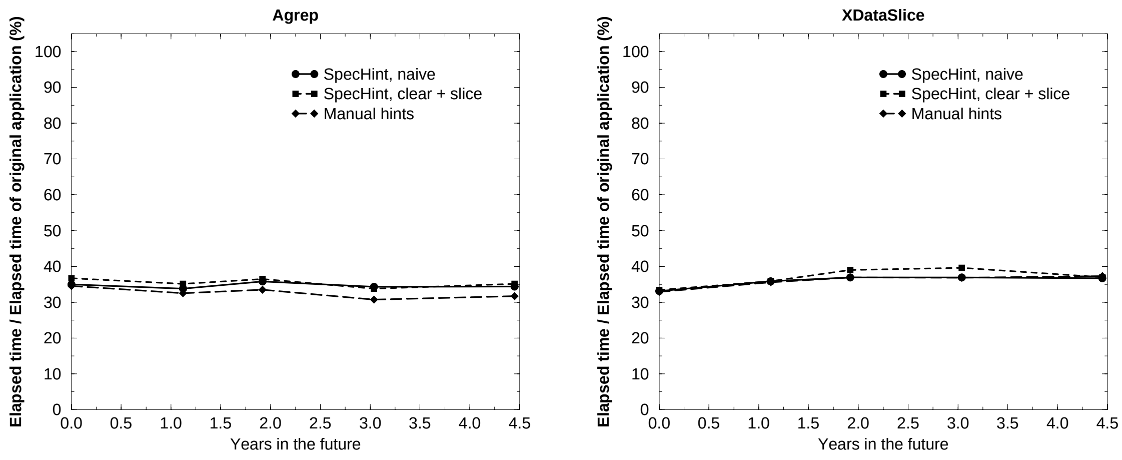

Figure 8.6: Projected performance of naive SpecHint (*SpecHint, naive*), SpecHint with logically clearing stale bits and experimental slicing (*SpecHint, clear + slice*), and manually modified applications (*Manual hints*) for Agrep and XDataSlice. The results for zero years in the future (i.e. now) were measured without any delay in the striper. The other results were obtained by adding delay in the striper as described in the text. The number of years that the simulated system is more advanced than the base system was calculated assuming a 58% annual increase in the processor speed, and a 8% annual increase in the disk access rate.

the system that the original request completed until that time. Finally, I assume that the relative values of the resulting elapsed times for original, speculating and manual hinting executables is indicative of what their relative elapsed times will be on future systems for which the gap between processing speeds and disk access times is n times that gap on my testbed.

This simulation is not fully accurate because it will misrepresent the relative speed of disk data transfer, which has been improving at a different rate than disk positioning times. In particular, disk data rates have been improving at 40% per year, so the simulation uses an artificially slow transfer rate. There are two factors which lessen the impact of this inaccuracy on the simulation results. First, most of the data requests are relatively small, so positioning time dominates the disk service time. Second, since the disks perform track-buffer read-ahead while the striper is delaying request initiation and completion, disk requests which are physically sequential will appear to have a faster than modelled transfer rate.

Figures 8.6 and 8.7 show results of simulating the future performance of naive SpecHint, SpecHint with the *Clear* and *Slice* mechanisms, and the manual hinting executables. Only the *Clear + slice* combination of the optional mechanisms was tested since the prior section found this to be the best combination.

Unsurprisingly, the performance benefit of the manual hinting executables increases, but only in proportion to the percentage of their elapsed times that was due to processing

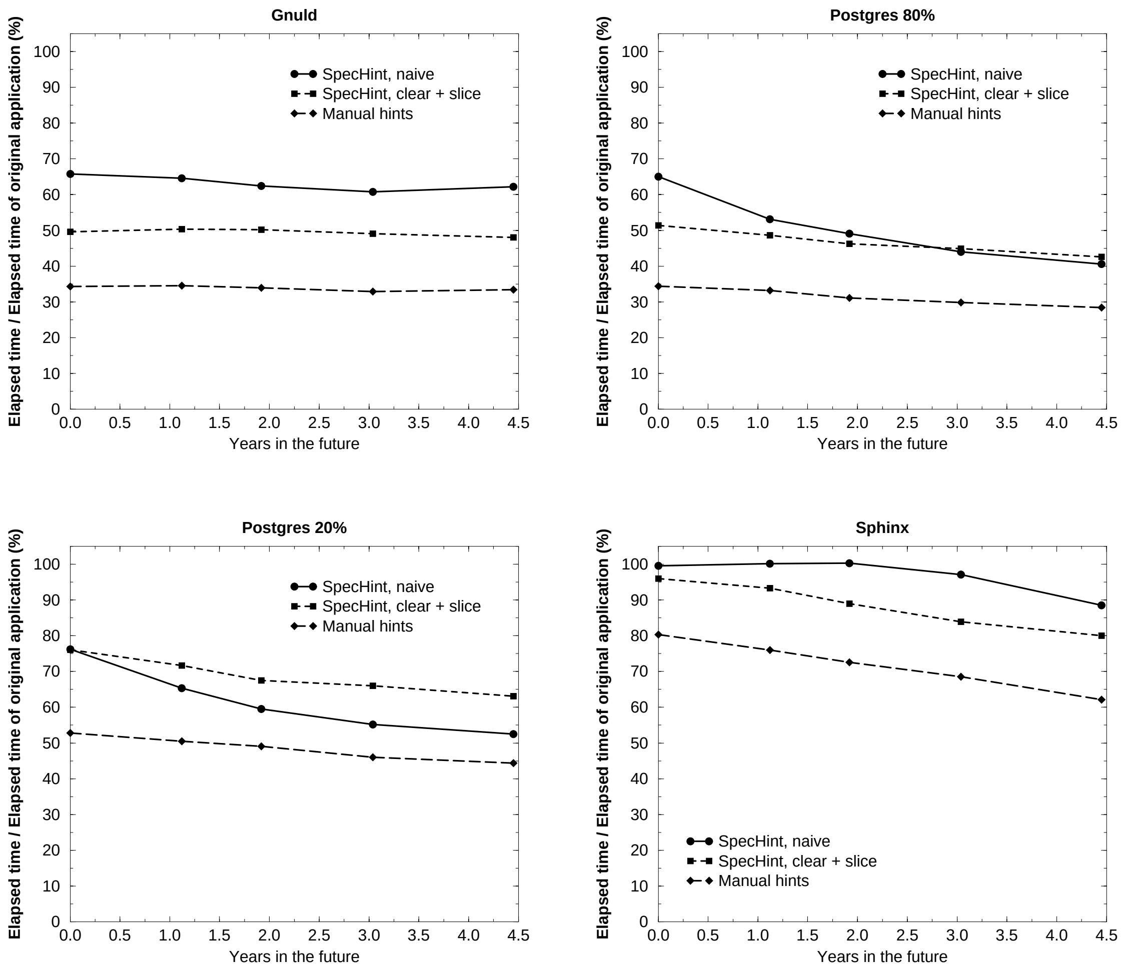

Figure 8.7: Projected performance of naive SpecHint (*SpecHint, naive*), SpecHint with logically clearing stale bits and experimental slicing (*SpecHint, clear + slice*), and manually modified applications (*Manual hints*) for Gnuld, Postgres 20%, Postgres 80% and Sphinx.

rather than I/O stall (which is largest for Sphinx, as indicated by the figures in Table 8.1). The curves for the speculating executables are similar in shape to those for the manual hinting executables for all but the Postgres benchmarks, for which the curves for the naive SpecHint executables converge towards the curves for the manual hinting executables. This indicates that, at least on systems in the range of the simulation experiment's projection, speculative execution may be able to leverage any increased abundance of spare processing cycles to deliver as or greater benefits than on my testbed. In particular, for the Postgres benchmarks, the convergence of the naive SpecHint curves towards the manual hinting curves indicate that speculative execution mainly needs more spare processing cycles. In addition, notice that, while SpecHint does not improve the performance of Sphinx by a substantial amount on my testbed, the simulation experiment indicates that the approach may yield substantial improvements for this benchmark on future systems with more spare processing cycles.

Finally, consider the relative performance of naive SpecHint and *Clear + slice*. For both Postgres benchmarks, the curves for naive SpecHint and *Clear + slice* cross. The reason is that, due to the incremental nature of the tests used to determine which chunks should be skipped, experimental slicing can erroneously identify a chunk of code as a chunk that should be skipped when the speculating thread sometimes needs to execute that chunk to generate correct hints. When spare cycles are plentiful, so there is less benefit to skipping chunks, these erroneous identifications are more likely to hurt the performance of speculative execution. On the other hand, for Gnuld, the relative performance of naive SpecHint and *Clear + Slice* remains the same, indicating that even an up to five times increase in the number of cycles per I/O stall is not sufficient for naive SpecHint to proceed past the expensive and unnecessary loops that experimental slicing allows the speculating thread to skip.

8.4 Concurrent applications

The single application test results described in the previous sections do not fully reveal the performance of the speculative execution approach since speculative execution relies on spare processing resources and increases contention for shared machine resources. Concurrent applications will reduce the number of spare processing cycles, which may hamper speculative execution's ability to generate prefetching hints. Moreover, by increasing the contention for shared machine resources, speculating applications may hurt the performance of concurrently executing applications. This section investigates these issues by examining the performance of speculative execution when there are fewer spare processing cycles, and the impact on application performance of replacing original, non-hinting executables with speculating executables in a multi-application mix. To isolate the impact of adding speculative execution as opposed to adding both speculative execution and prefetching, results are also shown for replacing non-hinting executables with manual hinting executables.

8.4.1 Impact of decreasing spare processing resources

To examine the effect of concurrent applications reducing the availability of spare processing resources, I conducted experiments in which a disk-bound application executes concurrently with a CPU-bound application. The CPU-bound application is a simple dummy application which sits in a tight loop, sleeping periodically. It executes at the default scheduling priority. Therefore, when executed with a speculating executable, the speculating thread will execute only when either both the CPU-bound application and the original thread in the speculating executable are blocked, or as the result of priority inversion due to the operating system's no-starvation scheduling policy (as discussed in Section 5.1.1).

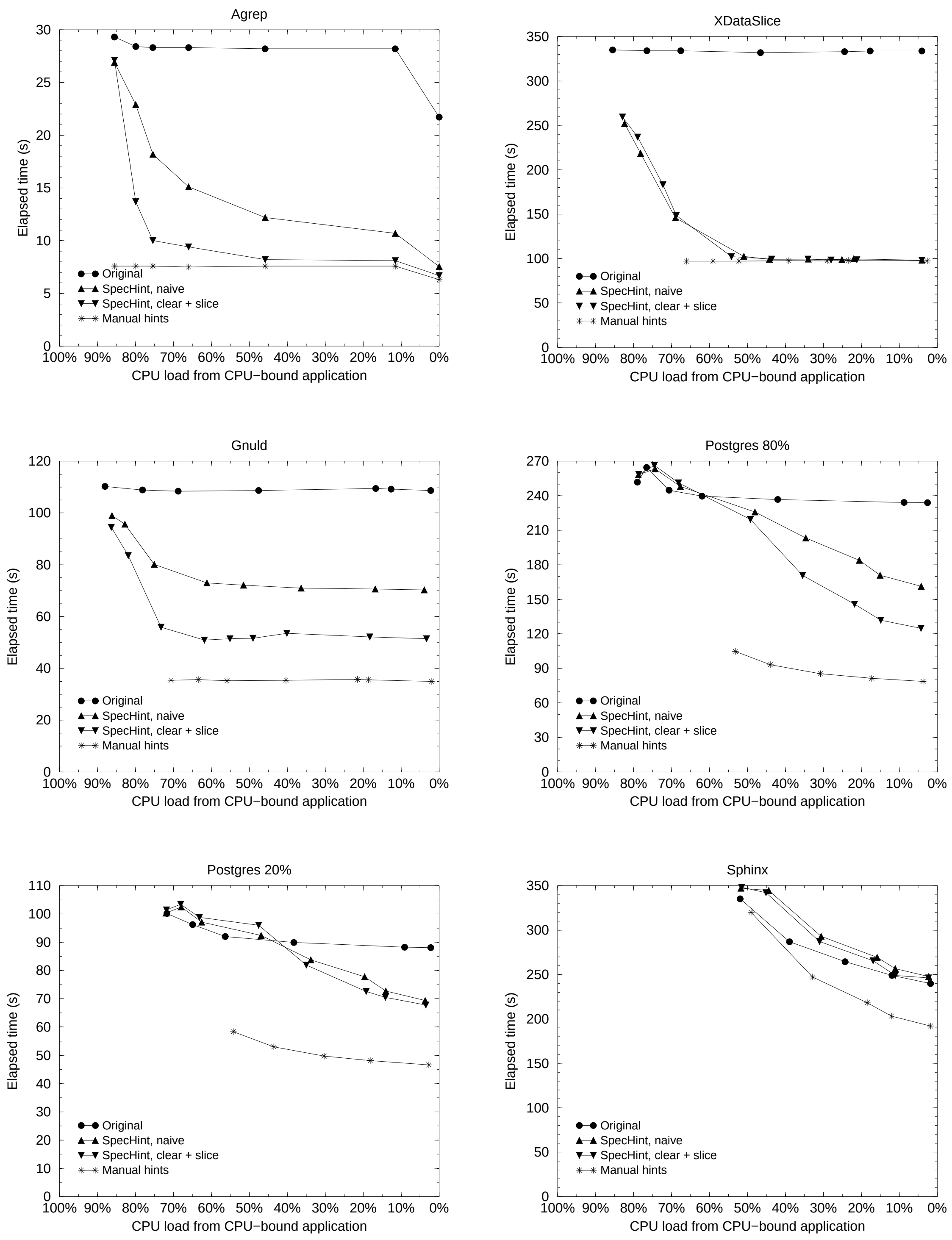

Figure 8.8: Elapsed time of the original, non-hinting application, the naively speculating applications, the speculating applications including the *Clear* and *Slice* mechanisms, and the manually modified applications when executed concurrently with a dummy CPU-intensive application. The file system is striped across four disks.

4 Disks						
	Agrep	XDataSlice	Gnuld	Postgres 80%	Postgres 20%	Sphinx
CPU load	10%	12%	9%	20%	29%	66%

Table 8.12: Percentage of elapsed time spent processing for original, non-hinting applications executed individually, on a four-disk system. These figures were calculated from the results in Table 8.1.

Figure 8.8 shows the results for this experiment. The y-axis in these graphs show the elapsed time of the disk-bound application. The x-axis shows the CPU load incurred by the CPU-bound application. Notice that the percentage of processing cycles which are spare for a data point with an x-value of $L_{CPU-bound}$ is less than $100\% - L_{CPU-bound}$; in particular, it is $(100\% - (L_{CPU-bound} + L_{Originalthread}))$ where $L_{Originalthread}$ is the CPU load incurred by the original thread of the benchmark application in obtaining that data point. For each line on each graph, the left-most data point shows the elapsed time when the CPU-bound application never sleeps; the amount by which the x-value of this data point is less than 100% is approximately the load incurred by the original thread of the benchmark application in obtaining this data point.

The results show that the elapsed time of the manual hinting executables, as well as that of the original, non-hinting executables, are not greatly affected by CPU load for all the benchmarks except Sphinx. This is not surprising since these executables do not rely on spare processing cycles to generate demand or prefetch I/Os, and Sphinx spends by far the most of its elapsed time processing (so it is inherently much more sensitive to competition for processing cycles). Table 8.12 shows the percentage of elapsed time spent processing for each of the original, non-hinting executables when they were executed alone on a four-disk system.

On the other hand, barring priority inversion due to the operating system's fairness policy, the speculating executables will not be able to generate prefetch I/Os unless there are spare processing cycles in which to perform speculative execution. This is indicated by the lack of improvement in the elapsed time of most of the speculating executables when the CPU-bound application never sleeps (the left-most data point in each graph). Notice that speculating XDataSlice and Gnuld are able to use even this minimal number of cycles to deliver substantial performance benefit.

The steep downward slope of the results curves for speculating Agrep, XDataSlice and Gnuld show that these speculating executables can provide significant performance benefits even at high CPU load. For example, even when only 5% of the processing cycles were available for speculative execution (because the CPU-bound application plus the original thread of the speculating executable consumed 95% of the processing cycles), naive SpecHint provides an 18% decrease in the elapsed time of the Agrep benchmark, while SpecHint with the *Clear* and *Slice* mechanisms provides more than a 50% decrease.

The more gradual downward slope of the results curves for the speculating Postgres benchmarks, and SpecHint's inability to improve the performance of the Sphinx benchmark on current systems, reflects the fact that these benchmarks are more CPU-intensive, as shown in Table 8.12. After all, the more cycles normal execution requires between `read` calls, the more cycles speculative execution is likely to require to generate prefetches. In the graphs, this can be seen in SpecHint's ability to provide benefit at higher CPU loads for the (less CPU-intensive) Postgres 80% benchmark than the (more CPU-intensive) Postgres 20% benchmark. Regardless, notice that speculating Postgres is able to provide substantial benefit even when there are substantially fewer processing cycles. This suggests that, while SpecHint would not improve the performance of Postgres if Postgres is executed concurrently with a CPU-intensive application, it could still deliver substantial improvements if Postgres is executed concurrently with, for example, less CPU-intensive interactive applications.

Finally, comparing the performance of the naive SpecHint and SpecHint with the *Clear* and *Slice* mechanisms shows that these mechanisms sometimes have a greater benefit than revealed by the single application results in Section 8.2. In particular, by decreasing the amount of processing cycles that the speculating thread requires to generate prefetches, they increase the benefit that speculative execution can provide when fewer processing cycles are available. Notice that, while these mechanisms did not provide benefit for the Agrep benchmark in single-application tests, they reveal a substantial benefit for the Agrep benchmark when there are fewer processing cycles.

8.4.2 Impact of increasing resource contention

To examine how increasing contention for I/O bandwidth and memory, as well as processing, resources will effect the performance of speculating applications, I ran experiments in which a pair of disk-bound applications were executed concurrently. Each graph in Figure 8.9 shows the results for a particular experiment (i.e. one pair of benchmarks). The benchmark pairings tested were: Agrep/Gnuld, Gnuld/Postgres 20%, Sphinx/Postgres 80%, and Postgres 80%/XDataSlice.

Concurrently executing two disk-bound applications could potentially improve application throughput, particularly when there is parallelism in the I/O system. While one application is stalled on I/O, the other application may be able to make use of the processor. Furthermore, if both applications are stalled on I/O, the I/O system may be able to service their requests in parallel. On the other hand, concurrent applications may compete for shared machine resources in a mutually harmful manner. For example, the interleaving of disk requests may hurt I/O performance by disrupting per-application request locality, increasing disk head positioning time. Comparing the total elapsed times for concurrently executed original, non-hinting binaries (as indicated by the left-most bar in each of the graphs in Figure 8.9) with the sum of the elapsed times measured in single application tests of the original, non-hinting executables (shown in Table 8.1) indicate that the former effect is more pronounced for these tests. In particular, the total elapsed time with concurrent original executables was greater than the sum of the elapsed times of the individual original executables in the single application tests only for the Postgres 80%/XDataSlice test. In all

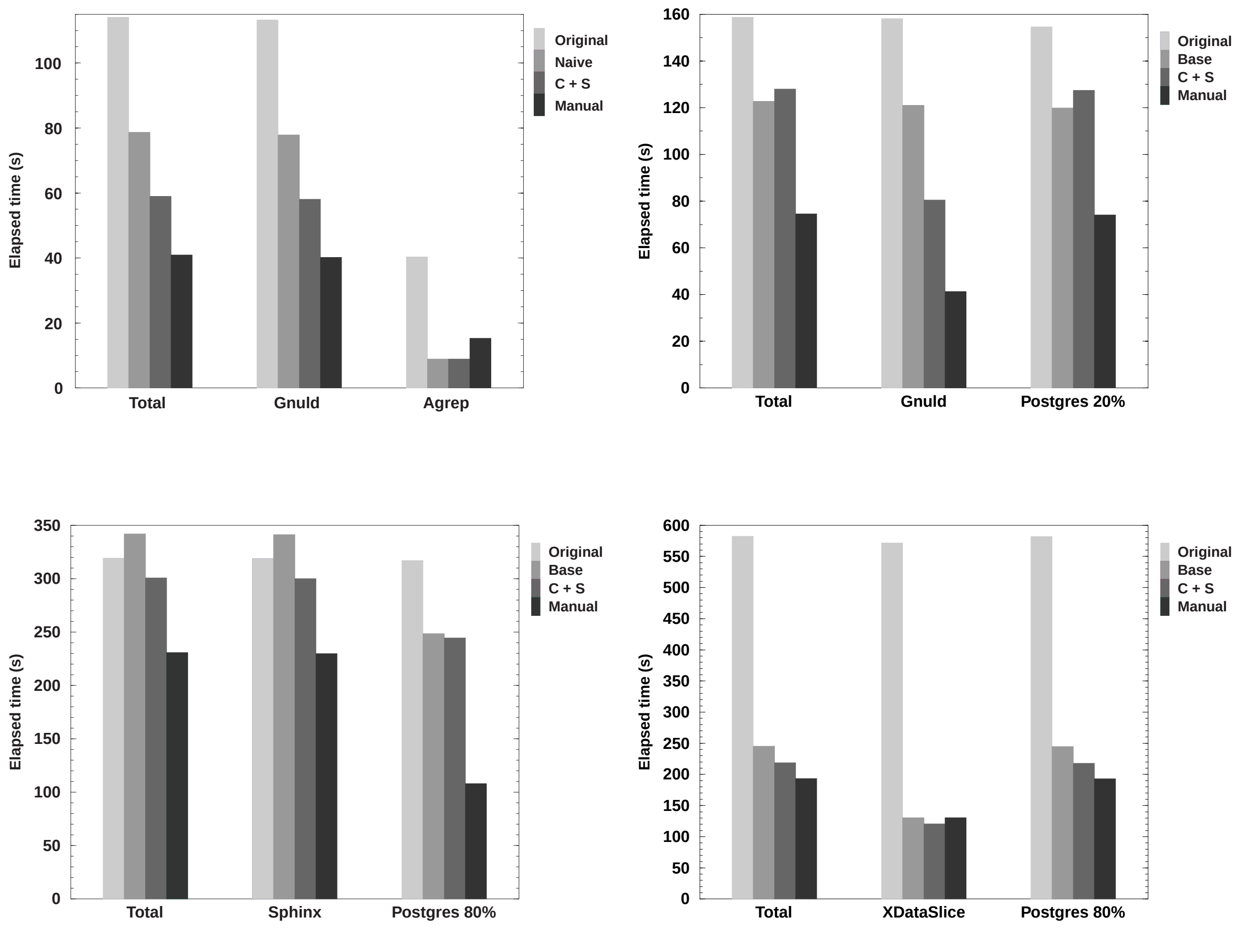

Figure 8.9: Elapsed time of multiple disk-bound applications when both applications are original, non-hinting applications, naive SpecHint applications, speculating applications with the *Clear* and *Slice* mechanisms, or applications manually modified to issue hints. The file system was striped across four disks.

the other combinations, the total elapsed time was less than the sum of the elapsed times of the individual original executables by a noticeable amount.

The elapsed times when the original, non-hinting executables were executed concurrently is a baseline from which the performance benefit of hinting executables can be seen. The elapsed times when the manual hinting executables were executed concurrently can be taken as the optimal performance with prefetching. When multiple speculating executables execute concurrently, the increased competition for I/O bandwidth between the prefetches on behalf of each executable will increase the sensitivity to hints which are issued later than they would be with manual hinting executables. In addition, the speculating threads must compete between themselves for a smaller pool of processing cycles (since the two original threads claim a larger percentage of the total processing cycles). Nevertheless, the results show that both naive SpecHint and SpecHint with the *Clear* and *Slice* mechanisms are able to provide substantial performance gains for all the applications except Sphinx in the Sphinx/Postgres 80% tests. *Clear + slice* improves the performance of Sphinx by only a small amount, while naive SpecHint degrades the performance of Sphinx by a small amount. This is not surprising since naive SpecHint is not effective for Sphinx even in the single application tests. Overall, however, these results are consistent with the results of the prior section, which showed that speculative execution does not require many cycles to be effective.

8.5 Transformation overhead

The SpecHint design requires that applications undergo a binary transformation step before they can take advantage of the speculative execution approach. The SpecHint tool takes a noticeable amount of time to transform an application binary because it performs several analysis and transformation steps (e.g. to create and add safety checks to shadow code). It also produces binaries that are substantially larger than the original binaries because they contain additional code and data structures to support speculative execution. This section demonstrates that the SpecHint design is feasible by showing that the cost of transforming the benchmark applications is not prohibitively large, even with the SpecHint tool, which is an unoptimized research implementation.

The figures in this section were obtained by transforming the application binaries using the SpecHint tool according to the SpecHint design on an AlphaStation 500 (500 MHz Alpha 21164 processor) with 1.5 GB of main memory. Table 8.13 shows the elapsed time for transforming each benchmark application. In particular, the SpecHint tool transformed the benchmark applications in 24 to 139 seconds. These are reasonable amounts of time since each application only needs to be transformed once. Moreover, while larger applications took longer to transform, notice that the relationship between size and time is roughly linear.

The table also shows the sizes of the original and transformed binaries. Since a non-research implementation would handle dynamic libraries, obviating the inclusion of the libraries for threading in transformed binaries, the table also shows the size of the original binary with thread libraries. Ignoring the increase due to the libraries for threading, we can see that the SpecHint tool increases binary sizes by an average of 174%. Increasing the sizes of binaries raises two concerns: whether there will be adequate storage for the larger binaries, and whether the performance of the binaries will degrade since an increase in binary size could hurt memory and processor cache performance. The latter concern can be dismissed because, as demonstrated in the preceeding sections, the transformed binaries almost always have better performance than the original binaries. In terms of storage overhead, the increase in binary sizes, while substantial, should not be prohibitive. Storage capacity is increasing at 60% a year. Indeed, the increasing abundance of storage

Benchmark	Transformation time (s)	Original size	Original + thread libraries size	Transformed size	Δ
`Agrep`	24	232 KB	864 KB	1.5 MB	632 KB
`XDataSlice`	139	4.4 MB	4.4 MB	8.9 MB	4.5 MB
`Gnuld`	30	536 KB	1.1 MB	2.1 MB	1.0 MB
`Postgres`	125	2.3 MB	3.0 MB	6.4 MB	3.4 MB
`Sphinx`	34	992 KB	1.5 MB	3.0 MB	1.5 MB

Benchmark	Increase in executable size due to				
	Support	Shadow code		Shadow data	
	routines	Unmodified copy	Safety checks	Read-only	Read-write
`Agrep`	10 KB	330 KB	97 KB	92 KB	103 KB
`XDataSlice`	10 KB	2.6 MB	905 KB	450 KB	574 KB
`Gnuld`	10 KB	532 KB	173 KB	158 KB	151 KB
`Postgres`	10 KB	1.8 MB	455 KB	428 KB	801 KB
`Sphinx`	10 KB	568 KB	166 KB	152 KB	640 KB

Table 8.13: Elapsed time for the SpecHint tool to transform each benchmark application according to the SpecHint design, and the amount by which it increases the size of executables. The size increase due to shadow code mainly consists of the copy of the instructions in the original code section.

has been the premise of several recent research projects [48, 58] that propose ways we can exploit this excess capacity.

The table also breaks down the increase in binary sizes. These figures show that, as intended, the SpecHint support routines comprise a very modest amount of code and data (10 KB). It also shows that, for all cases except Sphinx, over half of the increase is simply due to adding a copy of the text segment as the first step in setting up shadow code. Notice that the safety mechanisms described in Section 6.3 are a substantial contributor to code bloat. In particular, the combination of software copy-on-write checks, stack pointer checks, global pointer checks, and indirect control transfer target address checks account for 20% to 25% of the total size of shadow code. Most (around 80%) of this code bloat is due to adding software copy-on-write checks. The read-only data, which mainly consists of the data structures used by the indirect control transfer target address checks and the jump tables for shadow code, also account for a substantial portion of the size increase. Finally, the size increase due to read-write data is almost entirely unnecessary as it is just an artifact of some safe but inefficient assumptions that simplified my implementation (in particular, the way in which the SpecHint tool implements jump tables for shadow code creates shadow data copies of entire data sections from the original binary, rather than just the portions of those sections which contain jump tables).

8.6 Summary

This chapter presents experimental results that demonstrate that the SpecHint design is often effective, reducing elapsed times by 25% to 71% across five of six benchmarks on a four-disk system. The results also demonstrate that, for two of the six benchmarks, Agrep and XDataSlice, the SpecHint design can deliver near-optimal elapsed times (where applications manually modified to issue prefetching hints were assumed to yield optimal performance). For three other benchmarks, Gnuld and the Postgres benchmarks, the SpecHint design can deliver a substantial proportion of the potential performance benefits of application-level I/O prefetching.

For the four benchmarks for which the SpecHint design cannot deliver optimal performance, it sometimes falls short because the data values in read buffers of incomplete read calls are not available to it, while at other times it falls short because there are insufficient spare processing cycles in which to pursue speculative execution. An experimental slicing mechanism, particularly in combination with a simple single-value prediction mechanism, substantially increases the effectiveness of speculative execution for two of these four benchmarks, Gnuld and Postgres 80%, by enabling speculative execution to skip a substantial amount of unnecessary and (in Gnuld's case) misleading work. Moreover, simulation experiments projecting future performance indicate that the effectiveness of speculative execution may increase automatically for three of these four benchmarks, both Postgres benchmarks and the Sphinx benchmark, as the gap between processing speeds and disk access times continues widening.

The chapter also substantiates two basic claims about the SpecHint design: that it avoids increasing the amount of work performed during normal execution, and that it allows speculative execution to resynchronize with normal execution quickly. On the other hand, the evaluation demonstrates that the safety mechanisms in the design slow speculative execution by a substantial degree, and the design greatly increases memory pressure. Therefore, though no results were reported on systems that do not have ample memory, the design would probably degrade performance on such systems.

Speculative execution depends on the availability of spare processing and I/O resources. In both absolute and relative terms, SpecHint is much more successful on a system that supports I/O parallelism (i.e. by distributing data across more than one disk). On the other hand, SpecHint is able to provide significant performance benefits even on a single disk. Furthermore, multi-process experiments demonstrate that SpecHint can produce substantial performance gains even when there are far fewer spare processing cycles. These experiments also demonstrate that multiple speculating applications can produce substantial performance gains even when they compete with each other for processing, memory and I/O resources.

Finally, the chapter shows that the static transformation step required by the SpecHint design takes a reasonable amount of time, and produces binaries which, while larger, should not pose a storage problem when taking trends in storage capacity into account.

Chapter 9

Conclusions

The gap between processing speeds and I/O access times is widening. This trend is causing applications that must fetch data from disk to spend an increasing proportion of their execution time stalled on I/O, and to derive diminishing benefits from rapid increases in processor technology. I/O prefetching, a well-known technique for hiding disk latency, has the potential to dramatically decrease I/O stall time, particularly when the data that needs to be fetched is distributed across multiple disks. One major challenge to applying this technique in practice is the difficulty of generating accurate prefetches in a timely manner. This dissertation work focuses on the problem of generating prefetches without programmer involvement.

This chapter begins in Section 9.1 with a summary of the dissertation. Section 9.2 then highlights the contributions of this research. Finally, Section 9.3 discusses directions for future work.

9.1 Dissertation summary

Provided application source code is available, programmers could manually modify applications such that they would issue prefetch calls. However, such an approach can require substantial programming and debugging effort, and relies on the expensive resource of skilled programming labour. Techniques that automate the generation of prefetches are therefore desirable. Unfortunately, prior automatic techniques are insufficient for generic I/O-intensive applications with non-trivial access patterns. Motivated by this shortcoming, this dissertation work proposes a new automatic approach to initiating prefetching based on speculative execution, a technique that has long been used to avoid pipeline stalls in processors.

The key to the proposed approach is the unique mechanism it uses to predict what data a target process will access. In particular, it adds an execution of that process's code that exploits spare processing cycles. This added execution runs ahead of the target process's normal execution by skipping some operations, like blocking accesses to uncached data. This permits differences between the data values used during the added speculative execution and the data values that will be used during its target normal execution. Despite any such differences, the approach predicts that the data accesses encountered during a speculative execution will often be the same as the data accesses that will occur during its target normal execution. Thus, the approach predicts that, by initiating prefetching for that data, a speculative execution would be able to reduce the I/O stall time of its target normal execution.

This dissertation highlights three important goals to a design for adding speculative execution: safety, low overhead, and effectiveness. It discusses the challenges of guaranteeing safety, limiting overhead and promoting effectiveness. In each case, it discusses different possible design choices.

The dissertation also describes the choices taken within a prototype design and implementation of the proposed approach called SpecHint. The SpecHint design is a user-level design based on binary modification. It assumes operating system support for I/O prefetching, but no operating system support specific to speculative execution. For I/O prefetching support, my implementation of the SpecHint design takes advantage of the TIP informed prefetching and caching system, and a prefetch-aware software disk striper [43].

Two key elements of the SpecHint design are: 1) the addition of a new thread to each target process, and 2) the generation of a modified copy of the code from each original application binary, which is included in its transformed application binary. The added speculating threads are responsible for performing speculative execution on behalf of their process's normal execution. They accomplish this by executing the added shadow code, which is generated in a manner that prevents speculating threads from violating safety.

In order to quantify the effectiveness of the proposed approach, and the SpecHint design and implementation, and to evaluate the overhead in practice, I report on a variety of experiments using six benchmarks from the TIP benchmark suite. The benchmark applications are all real-world I/O-intensive applications, and include a wide variety of access patterns and application types. Two versions of each application's source code were used to produce the executables used in the evaluation. The original version of the source code, which does not contain prefetch calls, was used to produce both the base-line non-prefetching application executable, and the speculating executables. In addition, a version of the source code that had been manually modified to include prefetch calls was used to produce the manual prefetching executable.

The experimental results demonstrate that the speculative execution approach, and the SpecHint design and implementation, are often effective, reducing elapsed times by 25% to 71% across five of the six benchmarks on a four-disk system. The results also demonstrate that, for two of the six benchmarks, text search and data visualization, SpecHint delivers near-optimal elapsed times (where the manual prefetching executables were assumed to yield optimal performance). For three other benchmarks, linking and two database joins, SpecHint delivers a substantial proportion of the potential performance benefits of optimal prefetching. Finally, simulations suggest that, on future systems, SpecHint may be able to deliver substantial benefit for the remaining (CPU-intensive) benchmark, speech recognition, as well.

9.2 Contributions

This dissertation work introduces a novel approach to generating I/O prefetches automatically. It is the first exploration of automating speculative execution entirely in software. It is also the first complete design and evaluation of a system for using speculative execution to hide I/O latency. It makes several key contributions in terms of general findings, backed by specific results, which can be summarized as follows:

- Speculative execution can be performed to generate prefetches while guaranteeing safety (no erroneous changes to system behavior) for a large class of applications, without requiring special operating system support. The dissertation details the assumptions that establish which applications the prototype design can be applied to safely, which includes the five disparate applications in the benchmark suite.

- Speculative execution is capable of identifying a substantial amount of the uncached data that will be accessed by a wide range of I/O-intensive applications. Experimental evaluation of the prototype implementation demonstrate reductions in the percentage of requests for uncached, unprefetched blocks ranging from 68% to 98% across four disparate applications (five benchmarks).

- There are sufficient resources on current systems for speculative execution to provide substantial performance benefits. Experimental evaluation of the prototype implementation demonstrate reductions in elapsed times of up to 25%, 53%, and 71% on one-, two-, and four-disk systems, respectively.

- Assuming that the gap between processing speeds and disk access times continues to widen, simulation experiments indicate that, on future systems, the benefit of this approach would not decrease, and may even increase for some applications.

- Multi-process experiments demonstrate that speculative execution can provide substantial performance benefit even when there are fewer spare processing cycles, and more competition for memory and I/O resources.

Finally, in the process of this dissertation work, I implemented a tool which modifies application binaries such that they will perform speculative execution to generate I/O prefetches. This tool could be used to aid further investigations in this area.

9.3 Future work

Having demonstrated that speculative execution is a viable approach to automating I/O prefetching, this dissertation encourages several directions for future work.

One direction is to investigate ways to improve the effectiveness of an implementation based on binary modification. For example, the figures in Table 8.8 demonstrate that software copy-on-write increased the amount of work performed during speculative execution by a factor of 2 to 9. It may be possible to increase the effectiveness of the SpecHint implementation by leveraging more sophisticated static analysis to reduce the number of software copy-on-write checks. As another example, inspection of the code of the benchmark applications, and experiments with manually-configured slices, demonstrate that there is much room for improving the mechanisms in the SpecHint implementation for identifying and skipping unnecessary work, particularly unnecessary work that misleads speculative execution.

Another direction is to investigate how a mostly user-level design could benefit from some operating system support specific to speculative execution. In this dissertation work, I demonstrated that a design for automating the speculative execution approach can deliver substantial performance benefits and guarantee safety for a wide range of applications without any operating system support specific to speculative execution. As discussed in Section 6.2, however, simple operating system support could obviate the need for many of the assumptions that restrict (to however small a degree) the range of applications for which a user-level design like SpecHint guarantees safety. Moreover, a major weakness of the SpecHint design is that it could substantially hurt the performance of normal executions when memory resources are not abundant. As discussed in Section 5.1.2, this weakness was driven by the lack of appropriate mechanisms on current operating systems. With operating system support, however, a design might be able to contain its effect on memory performance.

In addition, simple operating system support may also enable this approach to be applied effectively to a much broader range of applications. In this dissertation work, I demonstrated that this approach is effective for applications that issue explicit file read calls. It remains to be seen whether this approach would also be effective for applications that generate I/O requests by accessing mapped files, or for applications that page heavily from swap space. In both cases, it would be helpful to have an efficient mechanism for detecting whether some page is in memory. One possibility would be a bit vector indicating which pages are in memory, set by the operating system and mapped into a process's address space [36]. With such a bit vector, it would be easy to extend the SpecHint design to accomodate both types of applications. In particular, copy-on-write checks could be leveraged to detect whether a memory access would refer to a page not in memory (and, in the first case, fall within an address range in which a file is mapped). If so, the speculating thread could issue a prefetch call for the appropriate data and return some stale value; if not, the speculating thread could access and use the data in its computations without blocking on I/O. Such a design would probably be effective for applications that generate I/O requests by accessing mapped files. Due to the prevalence of pointer-chasing code, however, more work may be required to develop a design that would be effective for applications that page heavily from swap space.

Another direction is to investigate the capabilities of an in-kernel design based on forking as sketched in Section 3.2.2. In addition to containing memory overhead, such a design could easily guarantee safety with no assumptions aside from the base safety assumption regarding shared resource usage (for the reasons discussed in Section 6.2.1). In addition, by enforcing safety within the operating system, such a design could guarantee safety with less impact on effectiveness (e.g. without endangering effectiveness for applications which dynamically generate code). Furthermore, such a design may require substantially less implementation effort than any design based on binary modification. On the other hand, since control only passed to the operating system in certain circumstances, some mechanisms (like experimental slicing) that can be easily incorporated in a user-level design may be harder to incorporate efficiently in an in-kernel design.

Finally, with or without operating system support, other directions include investigating whether this approach provides benefit to multi-threaded applications, how to best exploit multi-processors, and how to share resources amongst multiple speculative executions. In the first case, three possibilities would be to create a single speculative thread/process for each target process, to create one thread/process per thread in the target process, or to create a pool of threads/processes shared by the threads in the target process. In the second case, as discussed in Section 4.1.2, one possibility would be use multiple processes to more accurately and quickly, and with less cost, detect whether some work is necessary. The last case is briefly discussed in Section 4.1.4.

Bibliography

[1] S. Akyurek and K. Salem. Adaptive block rearrangement. In *Proceedings of the IEEE International Conference on Data Engineering*, April 1993.

[2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, October 1991.

[3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1995.

[4] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transaction on Computer Systems (TOCS)*, 14(4):311–343, 1996.

[5] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[6] K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, May 1993.

[7] Samya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, January 1998.

[8] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler inserted releases to manage physical memory intelligently. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[9] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, July 1997.

[10] R. J. Feiertag and E. I. Organisk. The multics input/output system. In *Proceedings of the 3rd ACM Symposium on Operating Systems Principles (SOSP)*, 1971.

[11] National Center for Supercomputing Applications. XDataSlice for the X Window System. http://www.ncsa.uiuc.edu/, 1989.

[12] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency control for high contention environments. *ACM Transactions on Database Systems (TODS)*, 17(2):304–345, June 1992.

[13] G.R. Ganger and M.F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the USENIX Winter 1997 Technical Conference*, January 1997.

[14] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems (TOCS)*, 5(1):77–92, February 1987.

[15] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Proceedings of the USENIX Winter 1995 Technical Conference*, January 1995.

[16] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.

[17] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, October 1995.

[18] Ed Growchowski. Ibm leadership in disk storage technology. http://www.storage.ibm.com/technolo/growchows/grocho01.htm, 2000.

[19] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[20] J.L. Hennessy and D.A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[22] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[23] *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*. 2001.

[24] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[25] Kim Korner. Intelligent caching for remote file service. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, 1990.

[26] David Kotz and Carla Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*, December 1991.

[27] T. Kroeger and Darrell Long. Predicting file system actions from prior events. In *Proceedings of the USENIX Winter 1996 Technical Conference*, January 1996.

[28] T. Kroeger and Darrell Long. The case for efficient file access pattern modeling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS)*, March 1999.

[29] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[30] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35–45, January 1990.

[31] Samuel Leffler, Marshall Kirk MucKusick, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley, 1989.

[32] H. Lei and Dan Duchamp. An analytical approach to file prefetching. In *Proceedings of the USENIX Winter 1997 Technical Conference*, January 1997.

[33] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, December 1996.

[34] Tara Madhyastha, Garth A. Gibson, and Christos Faloutsos. Informed prefetching of collective I/O requests. In *Proceedings of the ACM/IEEE SC99 Conference*, November 1999.

[35] M.K. McKusick, W.J. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, August 1984.

[36] Todd Mowry, Angela Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI),* October 1996.

[37] E.M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages (POPL)*, January 1981.

[38] Brent B. Welch Nelson, Michael N. and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[39] John K. Ousterhout, Herve Da Cost, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, December 1985.

[40] M.L. Palmer and S.B. Zdonik. FIDO: A cache that learns to fetch. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, September 1991.

[41] R. Hugo Patterson. *Informed prefetching and caching*. PhD thesis, Carnegie Mellon University, December 1997.

[42] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the 3rd IEEE International Conference on Parallel and Distributed Information Systems*, September 1994.

[43] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP),* December 1995.

[44] C. Reummler and John Wilkes. Disk shuffling. Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 1991.

[45] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop,* August 1997.

[46] Kyung Dong Ryu and Jeffrey K. Hollingsworth. Linger longer: Fine-grained cycle stealing for networks of workstations. In *Proceedings of the ACM/IEEE SC98 Conference*, 1998.

[47] K. Salem and Hector Garcia-Molina. Disk striping. In *Proceedings of the 2nd IEEE International Conference on Data Engineering,* 1986.

[48] Douglas S. Santry, Michael J. Feeley, Norma C. Hutchinson, Alistar C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[49] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[50] Richard L. Sites and Richard T. Witek. *Alpha AXP architecture reference manual, second edition*. Digital Press, Boston, MA, 1995.

[51] Michael D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, April 1991.

[52] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, June 1995.

[53] Inshik Song and Yookun Cho. Page prefetching based on fault history. In *Proceedings of the USENIX Mach III Symposium*, April 1993.

[54] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[55] C. Staelin and Hector Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report CS-TR-283-90, Princeton University, September 1990.

[56] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, February 1998.

[57] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[58] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[59] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[60] Carl Tait and Dan Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[61] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[62] Andrew Tomkins, R. Hugo Patterson, and Garth A. Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 1997.

[63] Kishor S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, 26(10):938–947, 1977.

[64] Uresh Vahalia. *UNIX internals - The new frontiers*. Prentice Hall, 1996.

[65] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.

[66] P. Vonsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software - Practice and Experience (SPE)*, 20(3):225–242, 1990.

[67] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

[68] Chenxi Wang, Jonathon Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 2000.

[69] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[70] M.J. Wolfe. More iteration space tiling. In *Proceedings of the IEEE Supercomputing Conference*, November 1989.

[71] S. Wu and U. Manber. AGREP - a fast approximate pattern-matching tool. In *Proceedings of the USENIX Winter 1992 Technical Conference*, January 1992.